修士論文

スケーラブルかつ高精度な Spectre 脆弱性の検出手法

吉田 昂太

23M30767

東京科学大学
情報理工学院
情報工学系
情報工学コース

指導教員　権藤 克彦

2025 年 1 月

概要

多くの CPU は、投機実行を悪用する Spectre 攻撃に対して脆弱である。この脆弱性に対処するため、直列化命令の挿入や Speculative Load Hardening (SLH) などの防御手法が提案されている。しかし、これらの手法をすべての条件分岐命令に適用し投機実行を完全に抑制する場合、非常に大きな実行時オーバーヘッドが発生するという問題がある。そこで、プログラム内の Spectre 攻撃に脆弱なコード片（Spectre Gadget）を特定し、部分的に投機的実行を抑制することで、実行時オーバーヘッドを削減する手法が提案されている。このアプローチでは、最低限のオーバーヘッドで Spectre 攻撃を防御するために、高精度かつ効率的に Gadget を特定できるツールが求められる。

既存の記号実行を用いた Gadget の検出手法は、通常の実行パスに加え、投機的実行パスも探索する必要があるため、状態空間が増大し、大規模なプログラムに対してスケールしない問題がある。一方でファジングを用いた Gadget の検出手法は、カバレッジ不足により一部の状態が探索されず Gadget が見逃される可能性がある。そこで本論文では、記号実行とファジングの解析手法を組み合わせることで、スケーラビリティと精度の両立を目指す Spectre Gadget の検出手法を提案する。記号実行とファジングにはそれぞれ利点と欠点が存在するが、両者を組み合わせることで、それぞれの欠点を補完し合い、スケーラビリティと精度を両立できると考える。記号実行では、ネストされた分岐予測ミスの回数を制限することで、投機的状態の探索範囲を縮小し、記号実行のスケーラビリティを向上させる。ファジングでは、記号実行で得られた解析結果を利用し、記号実行で探索されなかった状態を効率的に探索する。

提案手法のプロトタイプを実装し、広く利用されている暗号化ライブラリである OpenSSL からいくつかのプログラムを選択して実験を行った。その結果、いくつかの検体において記号実行のスケーラビリティが大幅に向上することを確認した。また、解析全体としては、既存手法よりも効率的に Spectre Gadget が検出された検体を確認した。

目次

図目次

表目次

第 1 章

はじめに

Spectre 攻撃 [15, 22, 24, 27] は、CPU の脆弱な投機実行を悪用して、被害者プロセスの秘密情報を漏洩させる攻撃手法である。投機実行とは、先行する命令の結果を待たずに、将来必要になる可能性が高い処理を事前に実行する最適化手法であり、CPU の性能向上に大きく貢献している。しかし、この最適化手法はほぼ全ての最新の CPU に実装されているため、Spectre 攻撃は Intel、AMD、ARM など、多くの CPU が標的となり、セキュリティ上の重大な懸念を引き起こしている。

Spectre-PHT [22] は Spectre 攻撃の一種であり、CPU の分岐予測による投機実行を悪用し、被害者のメモリ空間から秘密情報を盗み出す。たとえば、配列アクセスがインデックスの境界チェックによって保護されている場合でも、CPU の分岐予測器が分岐条件が満たされるとを誤って予測すると、分岐条件の評価が完了する前に配列アクセスを行う可能性がある。これにより、配列の境界外アクセスが発生し、秘密情報が読み取られる可能性がある。分岐予測が誤っていた場合、CPU はレジスタなどのアーキテクチャ状態をロールバックするが、キャッシュなどのマイクロアーキテクチャ状態はパフォーマンス上の理由からロールバックされない。Spectre 攻撃はこの特性を利用し、投機実行中に読み取った秘密情報をキャッシュ状態に反映させることで、後から秘密情報を復元する。

Spectre 脆弱性の多くは、Intel などのハードウェアベンダーによる修正が期待できない [17]。そのため、Spectre 攻撃に対する防御は、プログラム開発者がソフトウェアレベルで実施する必要がある。代表的な防御手法としては、直列化命令の挿入 [30] と Speculative Load Hardening (SLH) [11] が挙げられる。元のプログラムの全ての条件分岐命令に対して直列化命令を挿入することで、分岐予測による投機実行を抑制し、Spectre 攻撃を防ぐことができる。しかし、直列化命令以降の全てのメモリアクセス命令の投機実行が抑制されるため、Spectre 攻撃に関与しない安全なメモリアクセス命令の投機実行も抑制されてしまい、大きなオーバーヘッドが発生する。一方で、SLH は、分岐条件と投機実行による漏洩のリスクがある危険なメモリアクセス命令との間に新たなデータ依存関係を追加することで、誤った投機実行時の危険なメモリアクセス命令の実行を抑制する手法である。SLH は、直列化命令を挿入する方法と異なり、特定の危険なメモリアクセス命令にのみ作用するため、投機実行の抑制範囲を限定できる。その結果、パフォーマンスへの影響を最小限に抑えながら、Spectre 攻撃を防御することができる。

最も安全な防御手法は、全ての条件分岐命令に対してこれらの手法を適用し投機実行を抑制することである。しかし、既存研究 [11, 30, 44] では、このような保守的なアプローチを採用する場合、プログラムのパフォーマンスが大幅に低下することが報告されている。そこで、プログラム中から Spectre 攻撃に脆弱なコード辺 (Spectre Gadget) を検出し、これらの Gadget のみに防御手法を適用することで、オーバーヘッドを軽減する方法が提案されている。

Spectre Gadget の検出手法は、大きく静的解析手法 [4, 14, 43, 44] と動的解析手法 [20, 30, 31] に分類される。静的解析手法の一例として、記号実行を用いる手法 [14, 43] が提案されている。この手法では、記号実行において通常の実行パスに加え、分岐予測による投機的なパスも考慮して探索を行うことで Gadget を検出する。記号実行を用いることで、他のパス非依存な静的解析手法と比較して Gadget

の検出がより正確であり、動的解析手法よりも網羅的にプログラムを解析することが可能である。しかし、記号実行固有の問題として、状態数の爆発により、大規模なプログラムに対してはスケールしないという問題がある。一方、動的解析手法ではファジングを用いる手法 [20,30,31] が提案されている。この手法では、対象プログラムに計装を施すことで、ソフトウェアレベルで投機実行をシミュレートし、ファジングを用いて投機実行中に発生した境界外アクセスを検出する。ファジングを利用した手法は、記号実行と比較して大規模なプログラムに対してもスケールするという利点がある。しかし、ファジング固有の問題として、カバレッジ不足により Gadget が見逃される可能性がある。

そこで本研究では、スケーラビリティと精度を両立させる新しい Spectre Gadget の検出手法を提案する。提案手法は、記号実行フェーズとファジングフェーズの 2 つのフェーズで構成され、それぞれ異なる解析手法を用いることで Gadget を効率的に検出することを試みる。記号実行フェーズでは記号実行を用いて Gadget を検出するが、既存手法 [14, 43] とは異なり、ネストされた分岐予測ミスの回数に制限を設けている。CPU はネストされた分岐予測ミスによる投機実行が可能であり [28]、従来の記号実行手法ではこのような状態を網羅的に探索するため、分岐予測ミスのシミュレーションを深くネストして行う場合がある。しかし、先行研究 [30] によると、ネストされた分岐予測ミスによって悪用可能な Gadget の数は非常に限られていることが示されている。そこで本研究では、記号実行フェーズにおいてネストされた分岐予測ミスの回数を制限し、投機的状態の探索範囲を縮小し、記号実行のスケーラビリティを向上させる。ファジングフェーズでは、ファジングを用いて記号実行フェーズで探索されなかった投機的な状態 (ターゲット状態) を集中的にテストし、見逃された一部の Gadget を効率的に検出することを試みる。この際、記号実行フェーズで得られた解析結果を用いて、不要な投機的な状態の探索をなるべく回避することで、ファジングのスループットを向上させる。また、解析結果を元に、テストケースのターゲット状態への到達可能性を評価するための計装を行う。この評価結果を活用してシードのスケジューリングを最適化し、ターゲット状態に到達する可能性の高いシードを優先的に実行させる。

提案手法の有効性を評価するため、提案手法のプロトタイプを実装し評価を行った。評価には、一般的に広く利用されている暗号化ライブラリである OpenSSL [2] からいくつかのプログラムを選択しデータセットを作成した。予備実験の結果、いくつかの検体において記号実行のスケーラビリティが大幅に向上することを確認した。解析全体としては、既存手法よりも効率的に Spectre Gadget を検出した検体を確認した。

本研究の主な貢献は以下の通りである：

- 記号実行とファジングを組み合わせた新しい Spectre Gadget の検出手法を提案した。
- 記号実行の解析結果を活用し、ファジングで効率的に状態を探索する手法を提案した。
- 予備実験を通じて、ネストされた分岐予測ミスを抑制することで記号実行のスケーラビリティが向上することを確認した。
- 予備実験を通じて、既存手法と比較して、提案手法が特定の検体においてより効率的に Spectre Gadget を検出できることを確認した。

第 2 章

背景

2.1 投機実行

現代の CPU は、命令の実行を命令フェッチ、デコード、実行などの複数ステージに分割するパイプライン方式を採用しており、これにより前の命令が全ての処理を終えるのを待たずに次の命令の処理を開始することが可能となる。このように、複数の命令を並行して処理することで、CPU のスループットを大幅に向上させる。しかし、次に実行すべき命令が先行する命令の実行結果に依存している場合、CPU は次に実行する命令を判断できず、処理を一時的に停止せざるを得ない (Stall)。このような状況は制御ハザードと呼ばれ、CPU のパフォーマンスに大きな影響を与える。この問題を回避するため、CPU は先行する命令の結果を待たずに、将来的に必要となる可能性の高い命令を投機的に実行する。これは投機実行と呼ばれる最適化手法であり、ほぼ全ての最新 CPU に実装されている。

分岐予測は、この投機実行を応用した最適化手法の一つである。分岐命令に遭遇した際、CPU は過去の実行結果に基づいて分岐先を予測し、後続の命令を投機的に実行する。分岐予測による投機実行を図 2.1 に示す。分岐予測が成功すれば、CPU は Stall を回避して効率的に命令を処理できる。一方、分岐予測が失敗した場合、投機実行された命令の結果は全て破棄され、正しい分岐方向から実行を再開する必要がある。この予測には CPU の分岐予測器が使用され、分岐命令ごとに過去の分岐結果を記録し、それに基づいて予測を行う。

投機的に実行された命令とその結果は、CPU 内部の Reorder Buffer (ROB) という機構で管理される。ROB は、投機実行された命令が依存する他の命令の完了を待ちながら、命令の実行順序を維持する役割を持つ。そのため、投機実行可能な命令数は ROB のサイズによって制限されており、通常 200 命令程度のマイクロオペレーション (μOP) が上限とされる。分岐先が確定し、予測が正しいと判明した場合、投機的実行の結果はレジスタやメモリなどのアーキテクチャ状態に反映される。一方、予測が誤っていた場合、ROB 内の投機実行の結果は全て破棄され、誤った予測が行われた時点のアーキテクチャ状態までロールバックされる。

2.2 Side Channel Attack

コンピュータのシステムにおいて、channel とは情報を送信する可能性のある媒体のことを言う。channel には大きく分けて legitimate channel と incidental channel の 2 種類が存在する [3]。legitimate channel はシステムの設計者が情報送信用に意図的に設計したチャネルであり、イーサネット、共有メモリ、IPC ソケットなどがある。逆に incidental channel は、意図せずに設計されたチャネルであり、リソースの競合、キャッシュの状態、電力消費の変化などがある。更に、セキュリティ脅威モデルのコンテキストにおいて incidental channel は covert channel と side channel の 2 種類に分類される。covert channel は悪意のある送信者と受信者が意図的に情報の伝達を行うために利用される

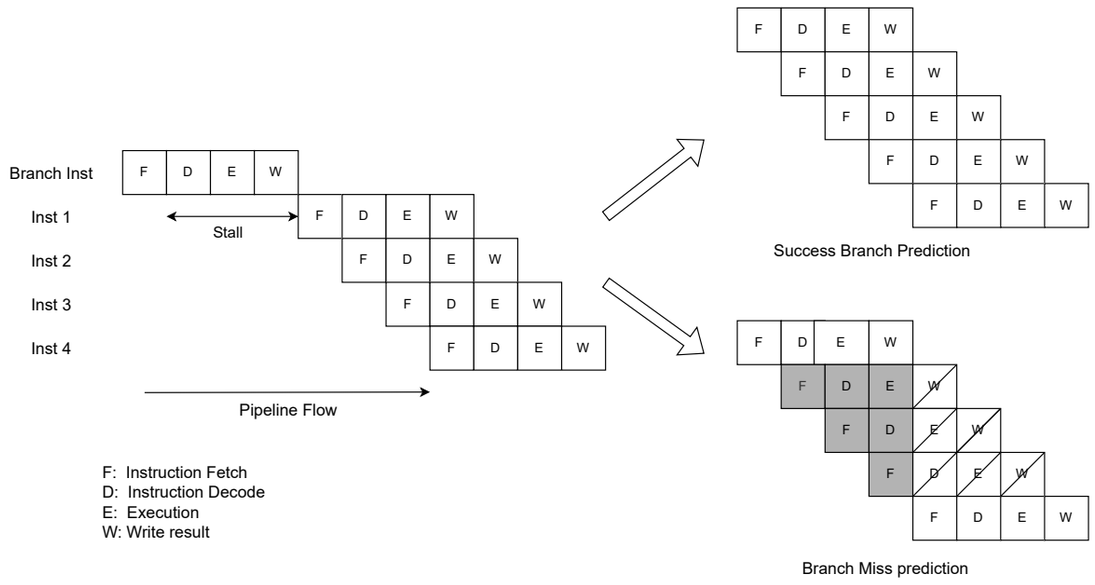

図 2.1: 分岐予測による投機実行

channel である。一方で side channel は、送信者は受信者に情報を伝達することを意図しておらず、情報が悪意のある受信者に伝達 (つまり漏洩) される際に利用される channel であり、このような channel を悪用する攻撃手法を side channel attack という。

side channel は 情報を伝達する方法に基づいて、Timing-based channel、Access-based channel、または Trace-based channel に分類される [38]。Timing-based channel は、様々な操作のタイミングを利用することで被害者の情報を推測する。例えば、攻撃者は様々な入力を暗号化または復号化し、その時の実行時間の差分を計測し分析することで、暗号鍵に関する情報を明らかにする [36]。Access-based channel は メモリやキャッシュなどの共有リソースに直接アクセスすることで、被害者のプロセスの情報を推測する。例えば、キャッシュがプロセス間で共有されていることを利用し、特定のキャッシュラインへのアクセス時間を計測することで、被害者プロセスの動作を推測する [26]。Trace-based channel は デバイスの消費電力や電磁放射などのプログラム実行時の詳細な情報を計測することで情報の漏洩を試みる。例えば、攻撃者は暗号化中のデバイスの消費電力を計測し分析することで、暗号化に関する情報を収集する [5]。

以降の節では、近年の CPU のキャッシュの基本的な構造と、それを side channel として利用する代表的な攻撃手法である Prime+Probe [39] と Flush+Reload [48] について説明する。これらの攻撃は本研究の対象である、Spectre 攻撃の中でも頻繁に利用される攻撃手法であるため、以降の節で詳しく説明する。

2.2.1 キャッシュ構造

キャッシュを side channel として利用する攻撃手法を紹介する前に、近年の CPU におけるキャッシュ構造について説明する。近年の CPU は、CPU とメモリの性能差に効率的に対処するため、複数の性能が異なるキャッシュメモリを階層的に配置する設計が採用されている。このキャッシュ階層は、CPU に近い順に L1 キャッシュ、L2 キャッシュと続き、最も遠いキャッシュは LLC（Last Level Cache) と呼ばれる。これらのキャッシュは CPU に近いほど高速だが容量が小さく、遠いほど容量は大きいが低速であるという特徴を持つ。一般的に L1 キャッシュと L2 キャッシュは各コア専用で、LLC は複数のコアで共有されているため、LLC は side channel attack の攻撃対象として適している。

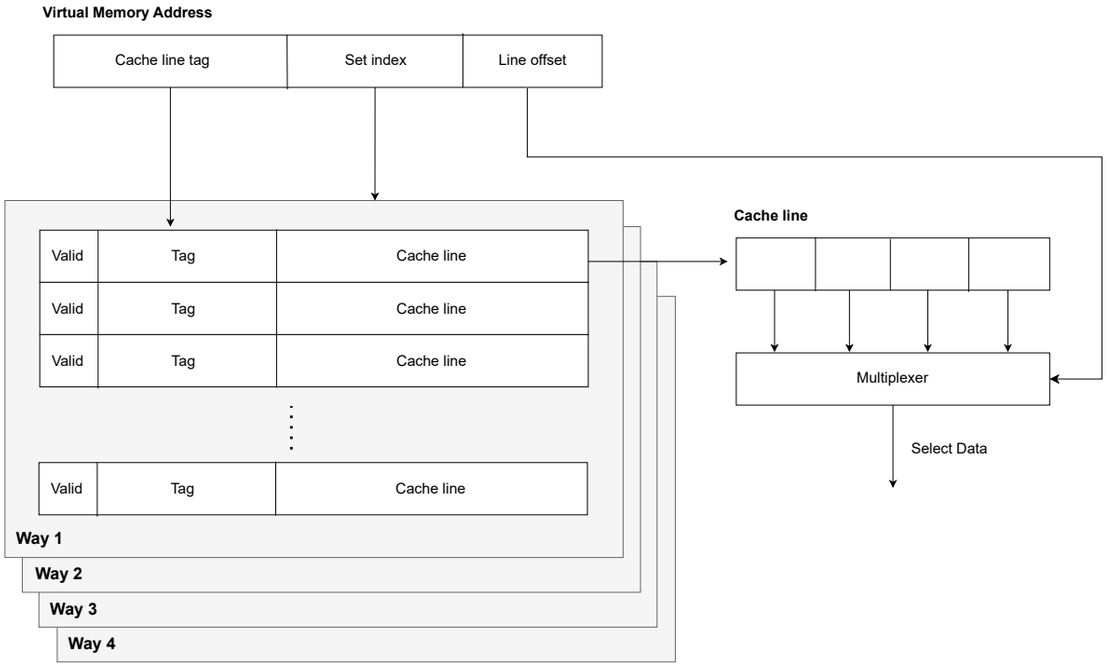

図 2.2: 仮想アドレスによるキャッシュの参照

キャッシュは複数のキャッシュセットで構成され、各セットには複数のキャッシュラインが含まれている。このようなキャッシュ構造をセットアソシエイティブ型と呼び、各キャッシュセット内のキャッシュライン数を連想度 (way) という。連想度が増加すると、キャッシュの競合性ミスが減少するが、比較器やキャッシュの設計コストが増加するため、バランスの取れた設計が求められる。

仮想メモリアドレスは、以下の 3 つの要素で構成される。

- Cache set index: データがどのキャッシュセットにマップされるかを決定する。
- Cache line tag: キャッシュセット内でデータを識別するタグ。
- Cache line offset: キャッシュライン内でどのデータワードが対象であるかを識別するタグ。

キャッシュは仮想アドレスの一部をインデックスとして使用し、各メモリアドレスを特定のキャッシュセットにマップする。そして、セット内の任意のキャッシュラインにデータを格納する。この際、マップ先のキャッシュセットがすでに埋まっている場合は、キャッシュ置換ポリシーに基づいて、どのキャッシュラインを削除するかが決定される。一般的に採用される手法は、Least Recently Used (LRU) や Least Frequently Used (LFU) といったポリシーである。仮想メモリアドレスによる 4way セットアソシエイティブ型キャッシュへの参照の概要を図 2.2 に示す。

2.2.2 Prime+Probe

Prime+Probe [39] はキャッシュを side channel として利用し、キャッシュ内の被害者のデータアクセスパターンから秘密情報を抽出する攻撃手法である。攻撃の概要について図 2.3 に示す。まず、攻撃者は、攻撃者プロセスにおけるコードブロックと被害者プロセスにおける機密性の高いコードブロックの両方がマップされるキャッシュセットを見つける。先述の通り、一部のキャッシュ階層は複数コア間で共有されているため、攻撃者と被害者プロセスが同一コアにスケジュールされていなくても攻撃することは可能である [26]。そして、攻撃者プロセスを一定時間実行することで、被害者と競合するキャッシュセットを自身のデータで埋める (Prime フェーズ)。その後、攻撃者は一定時間待機し、被害者プロ

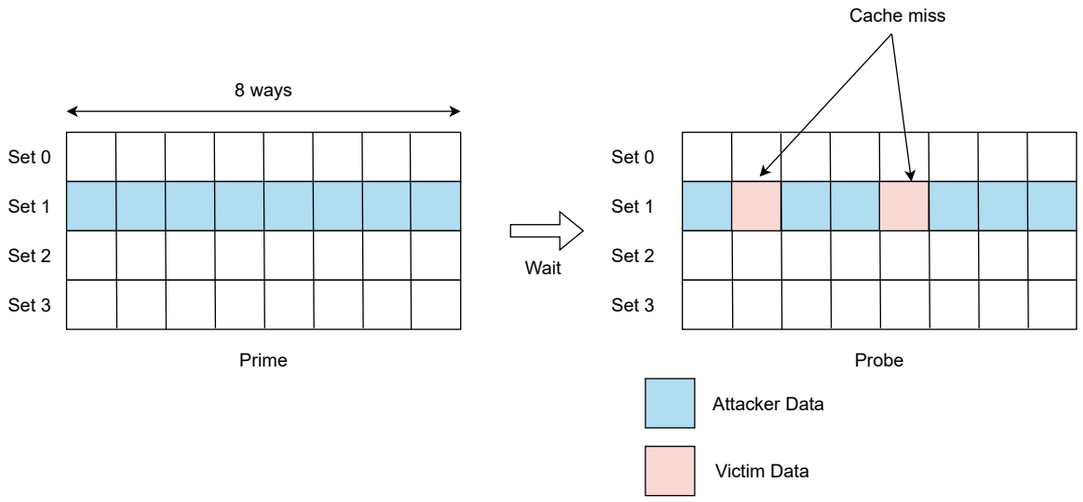

図 2.3: Prime+Probe の概要

セスが実行されるのを待つ。これにより、競合したキャッシュセットの一部は被害者のデータによって上書きされている可能性がある。その後、攻撃者は Prime フェーズで埋めたデータに再度アクセスし、アクセス時間を計測する (Probe フェーズ)。アクセス時間を計測することで、攻撃者のデータがキャッシュ上に存在するかがわかるため、待機時間の間にどのキャッシュラインを被害者プロセスがアクセスしたかがわかる。つまり、攻撃者は特定のキャッシュセットにおける被害者プロセスのメモリアクセスパターンを把握することが可能となる。

Prime+Probe は後述する Flush+Reload と比較して、フラッシュ命令や共有メモリを必要としないためより汎用的である。しかし、対象とするキャッシュが LLC のように大容量の場合、キャッシュセットを攻撃者のデータで埋めるまでに時間がかかるという欠点がある。

2.2.3 Flush+Reload

Flush+Reload [48] は Prime+Probe と同様に、データキャッシュを利用する side channel attack であるが、前提条件として被害者と攻撃者のプロセス間で特定のメモリ領域を共有している必要がある。プロセス間でメモリを共有するシナリオとして、プロセス間通信に利用される場合や、メモリフットプリントの削減のために、ハイパーバイザが仮想マシン間で同一内容のメモリページを共有する際に利用する場合がある [48]。

まず、攻撃者は共有されたメモリ領域の中から観測対象とするアドレスを選択し、clflush 命令などを使用して、選択したアドレスのキャッシュラインをキャッシュ階層全体から排除する (Flush フェーズ)。その後、攻撃者は一定時間待機し、被害者プロセスが観測対象のアドレスにアクセスするのを待つ。その後、攻撃者は観測対象のアドレスに再度アクセスし、アクセス時間を計測する (Reload フェーズ)。待機中に被害者プロセスが観測対象のアドレスにアクセスした場合、そのキャッシュラインはキャッシュで利用可能になり、アクセス時間は短くなる。一方、被害者プロセスが観測対象のアドレスにアクセスしていない場合、キャッシュラインをメモリから取得する必要があるため、アクセス時間は長くなる。このようにして、攻撃者は待機中に被害者プロセスが観測対象のアドレスにアクセスしたかどうかを把握することが可能となる。

Flush+Reload は攻撃のために前提条件が必要だが、Prime+Probe と比較して特定のアドレスのキャッシュラインをフラッシュすれば良いだけので、攻撃にかかる時間が短いという利点がある。

```
1 i = input();
2 if (i < array1_size) { // VB: Victim branch
3   secret = array1[i]; // RS: Read Secret
4   tmp &= array2[secret]; // LS: Leak Secret
5 }
```

図 2.4: Spectre-PHT 脆弱性を含むコード片

2.3 Transient Execution Attack

transient execution attack は、CPU の投機的実行によって一時的に実行される命令の結果がマイクロアーキテクチャに痕跡を残すことを利用した攻撃手法である。通常、CPU は誤った投機的実行が行われた場合、パイプラインをフラッシュし、アーキテクチャの状態を投機実行の前の状態までロールバックさせる。しかし、キャッシュなどの一部のマイクロアーキテクチャの状態は、パフォーマンスの観点からそのまま維持される。この特性を利用し、攻撃者は意図的に悪意のある投機的実行を誘発し、マイクロアーキテクチャの状態を介して秘密情報を復元することが可能となる。

transient execution attack は、2018 年に Spectre 攻撃 [22] と Meltdown 攻撃 [25] が明らかにされて以来、様々な CPU を標的とした新しい攻撃手法が次々と発見されてきた。これらの攻撃は大きく Spectre 型 [16, 22, 24, 27] と Meltdown 型 [25, 37, 40–42] に分類される [10]。Spectre 型は、データフローや制御フローにおける予測ミスによって発生する一時的な命令を悪用する。一方で、Meltdown 型攻撃は、例外を引き起こす命令に続く一時的な命令を利用する。

transient execution attack は大きく分けて 3 つのフェーズで構成される。まず攻撃者は分岐予測器やデータキャッシュの状態を調整し、マイクロアーキテクチャを特定の状態にする。次に、投機実行を引き起こすトリガー命令を実行する。これは、例外や分岐予測ミスなどを発生させることで、後続の命令が最終的に潰されるような命令である。CPU はトリガー命令が完了する前に後続の命令を一時的に実行する。この命令はマイクロアーキテクチャの side channel の送信側として機能し、秘密情報をキャッシュの状態に反映させたりする。トリガー命令の処理が終了すると、CPU は例外や分岐予測ミスを検知し、パイプラインをフラッシュし、アーキテクチャの状態をロールバックする。最後に攻撃者は、side channel の受信側で、メモリアクセス時間の計測を行うことで、マイクロアーキテクチャの状態から秘密情報を復元する。

2.4 Spectre 攻撃と対策

Spectre 攻撃 [15, 22, 24, 27] は CPU の脆弱な投機実行を悪用することで、被害者プロセスの秘密情報を漏洩させる攻撃手法である。投機実行は CPU のパフォーマンス向上に大きく寄与しており、ほぼ全ての最新の CPU に実装されている最適化手法である。そのため、Spectre 攻撃は、特定のベンターに限らず、ほぼ全ての CPU が攻撃対象となる可能性がある。2018 年に CPU の Spectre 脆弱性が発見されて以降、多くの対処法が研究されているが、未だに Spectre 脆弱性を完全かつ効率的に排除する手法は確立されていない。本章では、本研究の対象であり Spectre 脆弱性の一種である Spectre-PHT [22] と、その脆弱性に対する代表的な防御手法について紹介する。

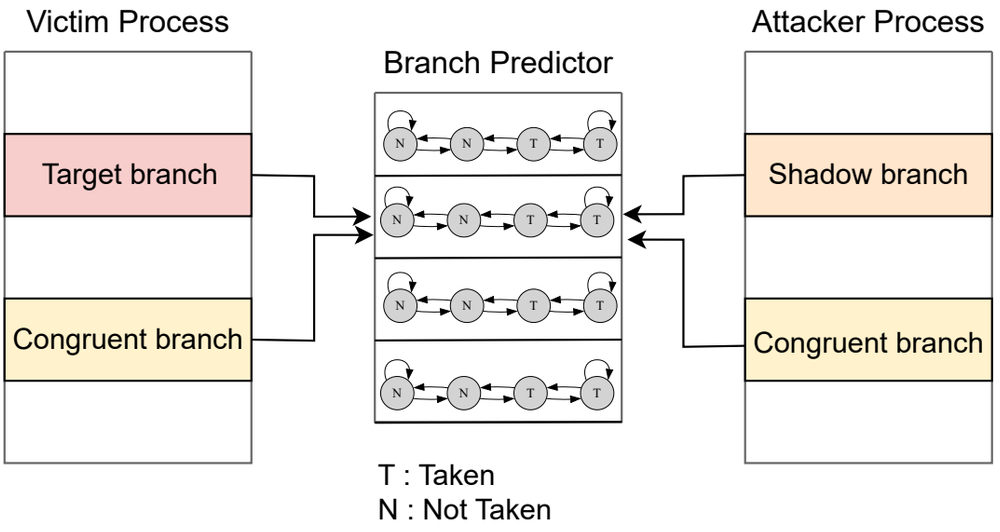

図 2.5: 分岐予測器のトレーニング

2.4.1 Spectre-PHT

Spectre-PHT（Spectre v1）[22] は Spectre 攻撃の一種であり、条件分岐命令の誤った予測によって引き起こされる投機実行による境界外へのメモリアクセスを利用する攻撃手法である。投機実行により条件分岐命令をバイパスするため、Bounds-Check-Bypass (BCB) 攻撃とも呼ばれる。

図 2.4 に示す Spectre-PHT 脆弱性を含むコード辺を用いて、具体的な攻撃手順を説明する。前提として、変数 `i` の値は外部から入力として与えられ、攻撃者が操作可能であるとする。

攻撃者はまず、2 行目の `if` 文 (Victim branch, VB) の条件が True と予測されるように分岐予測器をトレーニングする必要がある。近年の CPU における分岐予測器は、分岐命令の仮想アドレスをインデックスとして利用する [13, 22]。この特性を利用し、攻撃者は、Victim branch と同一の仮想アドレスを持つ分岐命令 (Shadow branch) を用いて、自身のプロセス内で分岐予測器をトレーニングできる。また、仮想アドレスの一部のみが分岐予測器のインデックスとして使用される場合には、Victim branch と仮想アドレスの一部が一致する分岐命令 (Congruent branch) を使用してトレーニングを行うことも可能である。これにより、攻撃者は図 2.5 に示すように、複数の経路を通じて分岐予測器をトレーニングできる [10]。図 2.4 のコード辺では、2 行目の条件分岐が True になるような入力を繰り返し与え、分岐予測器を True 側にトレーニングする。

次に、攻撃者は変数 `i` に `array1_size` 以上の値を与える。この場合、通常の実行では 2 行目の境界チェックによって、3 行目と 4 行目の処理は実行されずにプログラムは終了する。しかし、先述のトレーニングの結果、分岐予測器は分岐条件が True になると誤って予測するため、3,4 行目が投機実行される。この投機実行により、3 行目では攻撃者が操作可能な変数 `i` を使用して境界外アクセスが発生する (Read Secret, RS)。読み取られた値は、4 行目で `array2` へのアクセスのインデックスとして使用され、キャッシュの状態に変更が加えられる (Leak Secret, LS)。その後、CPU は分岐条件の評価結果から分岐予測が誤っていたことを知り、投機実行中のメモリアクセス結果を破棄するが、キャッシュの状態はパフォーマンスの観点からロールバックされない。

最後に、攻撃者は先述の Prime+Probe や Flush+Reload といった side channel attack を用いて、投機実行中のメモリアクセスで使用されたキャッシュラインを把握する。4 行目で読み取られたメモリ

```
1 #include <x86intrin.h>
2 i = input();
3 if (i < array1_size) {
4   _mm_lfence();   // Add lfence
5   secret = array1[i];
6   tmp &= array2[secret];
7 }
```

図 2.6: Spectre Gadget に対し、lfence 命令による防御策を適用した例

アドレスは秘密情報に依存しており、使用されるキャッシュラインの位置は仮想アドレスによって決定するため、どのキャッシュラインが使用されていたかを把握することで、秘密情報を復元することが可能になる。

以上のことから、Spectre-PHT 脆弱性は、(i) 攻撃者が制御可能な分岐命令 (VB)、(ii) 秘密情報を読み取る命令 (RS)、(iii) 読み取った秘密情報をキャッシュ状態に反映させる命令 (LS) の 3 つの命令から構成される。ただし、実際の攻撃では、VB に該当する命令の分岐予測から開始した投機実行が終了する前に、LS 及び RS に該当する命令の実行を完了する必要がある。これは、キャッシュ状態に秘密情報を反映させる前に投機実行が終了してしまうと、実行状態がロールバックされ、攻撃が成立しないためである。投機実行可能な命令数は CPU の ROB のサイズに制限されており (詳しくは 2.1 を参照)、投機実行可能な命令数の上限を Speculative execution window（投機ウィンドウ）という。このように、Spectre-PHT 脆弱性を含んだコード辺は Spectre Gadget と呼ばれる。以降では Spectre-PHT を単に Spectre と表記する。

2.4.2 直列化命令

Spectre 攻撃には投機実行が必要である。そのため、命令がその命令に至る制御フローが確定した場合にのみ実行されるようにすることで、Spectre 攻撃を防御することが可能である。このアプローチとして、元のプログラムに直列化命令を挿入して修正する方法が提案されている [22]。例えば x86 アーキテクチャの場合、lfence 命令を使用することが可能である。lfence 命令は、全ての先行するメモリロード命令が完了するまで、後続のメモリロード命令を投機実行させないようにする制御命令である。プログラム中の全ての条件分岐命令の分岐先に lfence 命令を挿入することで、それ以降のメモリロード命令の投機実行を抑制し、Spectre 攻撃を防御できる。

図 2.4 のコード片に対して lfence 命令を挿入して Spectre 攻撃に対して堅牢化したコード辺を図 2.6 に示す。4 行目に lfence 命令が挿入されることで、3 行目の分岐条件の評価が終了し分岐先が確定してから、後続の配列へのアクセスが行われるようになる。

しかし、CPU の分岐予測を完全に無効化することで、大幅にパフォーマンスが低下することが知られている。既存研究 [44] では、元のプログラムの全ての条件分岐命令に対して lfence 命令を挿入した場合、プログラムの実行時間が最大 3.25 倍程度に増加することが報告されている。

2.4.3 Speculative Load Hardening

Spectre 攻撃に対する実行時オーバーヘッドが少ない防御策として、Carruth によって提案された Speculative Load Hardening（SLH）[11] がある。SLH はコンパイラベースの防御手法であり、分岐命令を使用せずに、メモリアクセス命令が有効な制御フローパス上で実行されているかを確認するコー

```
1  void leak(int data);
2  void example(int* pointer1, int* pointer2) {
3    if (condition) {
4      // ... lots of code ...
5      leak(*pointer1);
6    } else {
7      // ... more code ...
8      leak(*pointer2);
9    }
10 }
```

図 2.7: SLH 適用前の Spectre Gadget

```
1  uintptr_t all_ones_mask = std::numerical_limits<uintptr_t>::max();
2  uintptr_t all_zeros_mask = 0;
3  void leak(int data);
4  void example(int* pointer1, int* pointer2) {
5    uintptr_t predicate_state = all_ones_mask;
6    if (condition) {
7      predicate_state = !condition ? all_zeros_mask : predicate_state;
8      // ... lots of code ...
9      pointer1 &= predicate_state;
10     leak(*pointer1);
11   } else {
12     predicate_state = condition ? all_zeros_mask : predicate_state;
13     // ... more code ...
14     int value2 = *pointer2 & predicate_state;
15     leak(value2);
16   }
17 }
```

図 2.8: SLH 適用後のコード辺

ドを計装する。以降では、先行研究 [11] に示されているコード例を元に、SLH の概要を説明する。

図 2.7 と図 2.8 は、それぞれ SLH 適用前と後のコード辺である。ここで、関数 `leak` は投機実行された場合に引数のデータを攻撃者に漏洩させると仮定する。変数 `predicate_state` は、現在の実行が分岐予測ミスによる分岐先であるかを表しており、7 行目と 12 行目において、条件 `condition` が満たされない場合は `all_zeros_mask`(全てのビットが 0)、満たしている場合は `all_ones_mask`(全てのビットが 1) が代入される。重要な点として、7 行目の三項演算子は分岐命令が使用されない形で機械語に変換される必要がある。これにより、7 行目の条件 `condition` が投機実行によってバイパスされないことを保証する。x86 アーキテクチャでは、cmov 命令を用いることで実現可能である [11]。

分岐予測が正しい場合は、`predicate_state` には `all_ones_mask` が格納される。そのため、9 行目及び 13 行目でビット論理積が取られても、`pointer1` 及び `pointer2` の値はそのまま保持される。一方、分岐予測が誤っている場合、`predicate_state` には `all_zeros_mask` が格納されている。これに

より、ビット論理積を取ることで、pointer1 と pointer2 の値は 0 となる。その結果、関数 leak が実行されても攻撃者が意図したデータは漏洩しない。

lfence 命令を用いた防御策では、lfence 命令以降に続く全てのメモリロード命令の投機実行が抑制される。このため、Spectre 攻撃に関与しない安全なメモリロード命令であっても、投機実行が制限されてしまう。また、分岐予測が正しく行われた場合でも、lfence 命令以降の投機実行が抑制されるため、パフォーマンスが大幅に低下する。一方、SLH は、投機実行による漏洩のリスクがある危険なメモリロード命令に対してのみ投機実行を抑制する。そのため、投機実行可能なメモリロード命令を増やすことができ、lfence 命令と比較して小さい実行時オーバーヘッドで Spectre 攻撃を防御することができる。しかし、それでも全てのメモリロード命令を SLH で強化する場合、大規模なアプリケーションでは実行時オーバーヘッドが 36% 程度になることが報告されている [11]。

2.5 Spectre Gadget の検出

Spectre 攻撃に対してプログラムを堅牢化するために、lfence 命令や SLH を全ての条件分岐命令やメモリロード命令に適用する場合、大きな実行時オーバーヘッドが発生する。そこで、プログラム内の潜在的な Spectre Gadget を検出し、これらの Gaget のみに防御策を適用して、実行時オーバーヘッドを軽減する方法が提案されている。

Spectre Gadget を検出するためのプログラム解析手法は大きく分けて、静的解析 [4, 14, 43, 44] による手法と、動的解析 [20, 30, 31] による手法の 2 種類に分類される。投機実行はハードウェアによる機能であるため、これらの解析手法の大半がソフトウェアレベルで投機実行をシミュレートすることで Spectre Gadget を検出している。以降では、それぞれの手法がどのように投機実行をシミュレートし、Spectre Gadget を検出しているかを説明する。その代表例として、静的解析による検出手法として KLEESpectre [43] を、動的解析による検出手法として SpecFuzz [30] を紹介する。これらの既存手法は本研究における実装の基盤にもなっているため詳しく説明する。

2.5.1 KLEESpectre

KLEESpectre [43] は記号実行により、プログラム中の Spectre Gadget を検出するツールである。記号実行とは、プログラムに具体的な値ではなく、記号的な変数を入力として与えることで、プログラムの全ての実行パスを網羅的に解析する静的手法である。入力を記号化することで、プログラムが異なる入力で取る可能性のある複数のパスを同時に探索することができる。記号実行は記号実行エンジンによって行われ、探索された制御フローパスごとに、(i) そのパスに沿って実行された分岐によって満たされる条件を記述するパス制約、および (ii) 各変数を記号式または具体的な値にマップする記号的なメモリ状態を保持することで現在の実行状態を管理する [7]。

KLEESpectre は記号実行エンジンである KLEE [9] を Spectre Gadget の検出用に拡張したものであり、単純なパターンマッチングによる手法 [4] と比較して高い精度で Gadget を検出することができる。

まず、[43] に挙げられるコード辺を用いて、KLEESpectre がどのように分岐予測による投機実行をシミュレートしているかを説明する。図 2.9 は典型的な Spectre 脆弱性を含んでいるコード辺である。分岐 b1 が誤って分岐予測された場合、x の値は SIZE 以上となるので、8 行目で境界外アクセスが発生し、秘密情報が temp に読み取られる可能性がある。その後、9 行目で temp が array2 へのアクセスのインデックスとして使用されることで、秘密情報がキャッシュの状態として漏洩する。

通常の記号実行の場合、条件分岐命令に遭遇すると、分岐条件を満たし、Taken 側の基本ブロックに進んだ状態と、分岐条件を満たさず、Not taken 側の基本ブロックに進んだ状態の 2 つの状態が新しく

```
1  uint32_t SIZE = 16;
2  uint8_t array1[16], array2[256*64], array3[16];
3
4  uint8_t foo(uint32_t x) {
5    uint8_t temp = 0;
6    if(x < size) {    // b1
7      // A
8      temp = array1[x];
9      temp |= array2[temp];
10     if(x <= 8) {  // b2
11       // B
12       temp |= array2[8];
13     }
14   }
15   // C
16   temp |= array3[8]
17   return temp;
18 }
```

図 2.9: KLEESpectre におけるコード例

生成される。しかし、KLEESpectre では投機実行をシミュレートするため、上記の 2 つの状態に加え、分岐条件が満たし、Not taken 側の基本ブロックに進んだ状態と、分岐条件を満たさず、Taken 側の基本ブロックに進んだ状態の 2 つの状態も生成される。

例えば、KLEESpectre が図 2.9 の分岐 `b1` に遭遇した場合、以下の 4 つの状態が新しく生成される。

(1) `x < SIZE` を満たし、分岐 `b1` が正しく分岐予測された状態
(2) `x < SIZE` を満たさず、分岐 `b1` が正しく分岐予測された状態
(3) `x < SIZE` を満たし、分岐 `b1` が誤って分岐予測された状態
(4) `x < SIZE` を満たさず、分岐 `b1` が誤って分岐予測された状態

(1) の場合、KLEESpectre は現在の状態のパス制約に `x < SIZE` を追加し、基本ブロック `A` に移動し、実行を続ける。(2) の場合も同様に、現在の状態のパス制約に `x >= SIZE` を追加し、基本ブロック `C` に移動し、実行を続ける。(3) の場合、KLEESpectre は現在の状態のパス制約に `x < SIZE` を追加するが、分岐予測ミスにより、基本ブロック `C` に移動し、実行を続ける。(4) の場合も同様に、現在の状態のパス制約に `x >= SIZE` を追加するが、分岐予測ミスにより、基本ブロック `A` に移動し、実行を続ける。このように KLEESpectre は分岐予測ミスによる投機的なパスも考慮して記号実行を行うように KLEE を拡張している。図 2.10 において、投機的なパスも含めた図 2.9 の制御フローグラフを示す。

図 2.10 では、実線が直前の分岐命令が正しく分岐予測された場合のパスを、点線が直前の分岐命令が誤って分岐予測された場合の投機的なパスを表している。緑色の基本ブロックは通常の実行パスにおける基本ブロックを、赤色の基本ブロックは投機的なパスにおける基本ブロックを表している。KLEESpectre は全ての分岐命令が攻撃者によって訓練されていると仮定し、投機実行のシミュレートを行う。このシミュレーションにおいて、投機的な状態が以下のいずれかの条件を満たした場合、CPU における投機実行の終了動作を再現するためにその状態を破棄する。

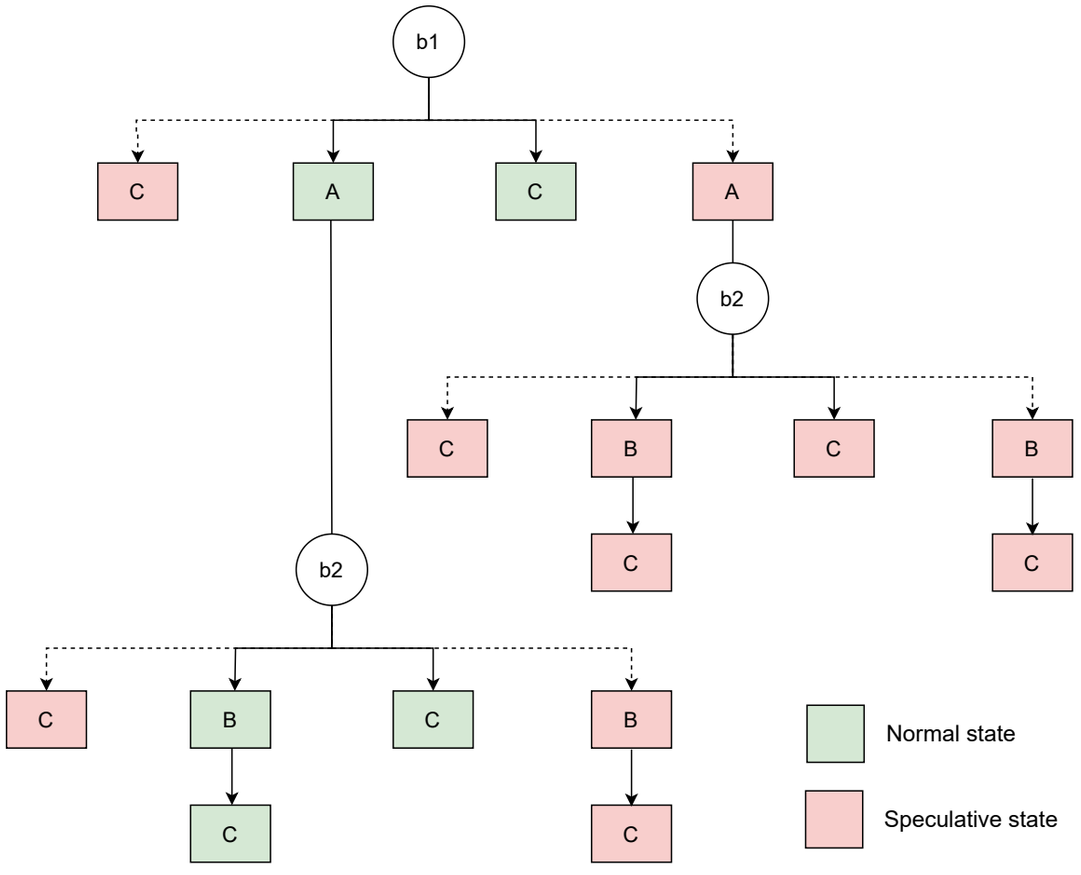

図 2.10: 投機的なパスを含んだ制御フローグラフ

- 投機的なパス上で実行した命令数が投機ウィンドウの制限に達した場合
- 直列化命令に遭遇した場合
- 例外が発生した場合

また、本研究のテーマと関連する重要な点として、KLEESpectre では分岐予測ミスをネストしてシミュレートする場合がある。図 2.9 では、分岐 `b1` の分岐予測ミスによる投機実行中に分岐 `b2` に遭遇した場合、KLEESpectre は重ねて分岐 `b2` の分岐予測ミスをシミュレートする。このように、KLEESpectre では、ある分岐命令の分岐予測ミスから開始された投機実行中に、別の分岐命令に遭遇した場合、重ねて分岐予測ミスをシミュレートする。実際の CPU においても、このようなネストされた分岐予測ミスによる投機実行は可能であり [28]、KLEESpectre は正しく CPU の投機実行をシミュレートしていると言える。

次に、KLEESpectre がどのように Spectre Gadget を検出するかについて説明する。まず、投機的に実行されたパスにおけるメモリアクセスを監視し、それらが秘密情報を参照している場合、その命令を RS（Read Secret）として記録する。KLEESpectre では、投機的パスにおいて攻撃者が操作可能な値 (つまり記号変数) をアドレスとして使用した境界外のメモリアクセスは、すべて秘密情報を参照していると仮定し、保守的な解析を行っている。境界外のメモリアクセスは KLEE に組み込まれているチェック機構を利用して、識別している。次に、秘密情報に依存する値を用いてメモリアクセスが行われた場合、その命令を LS（Leak Secret）として記録する。LS が検出されると、直前に分岐予測ミスが発生した分岐命令と、RS および LS に該当する命令のセットをまとめて、Spectre Gadget として記録する。

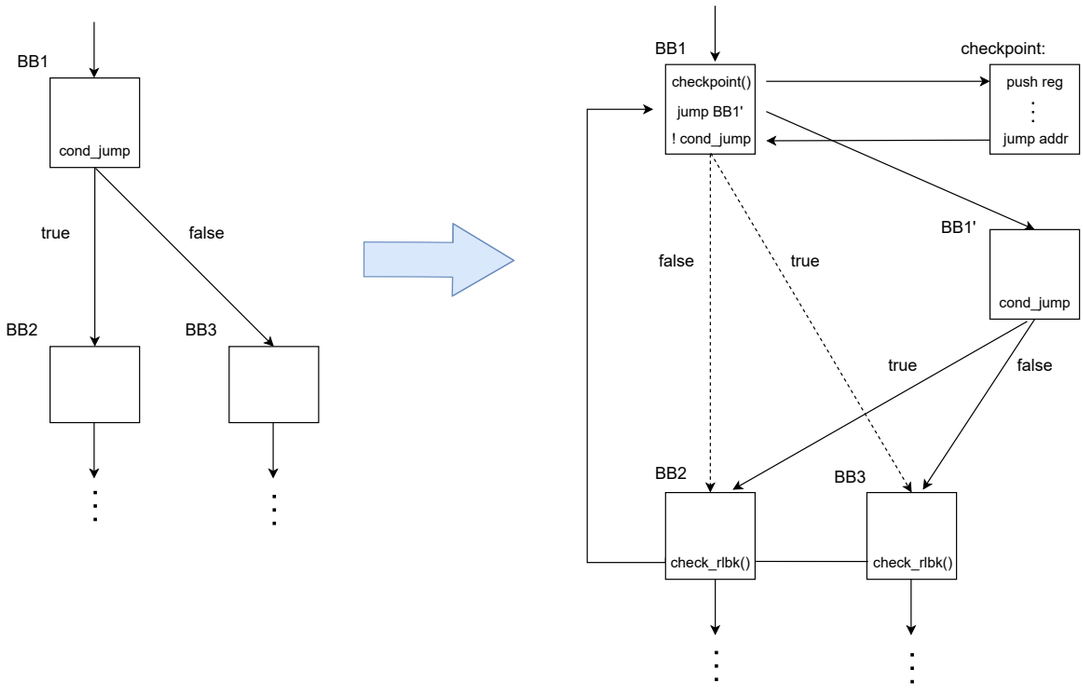

図 2.11: SpecFuzz による計装前と計装後の制御フローグラフの概略

KLEESpectre は記号実行を活用することで、単純な静的解析手法に比べて正確であり、動的解析手法に比べてプログラムの網羅的な解析が可能である。これにより、より高い精度で Spectre Gadget を検出できる。しかし、大規模なコードや探索すべきパスが多い複雑なプログラムに対しては、スケールしないという問題がある。

2.5.2 SpecFuzz

SpecFuzz [30] は、ファジングを利用してプログラム内の Spectre Gadget を検出するためのツールである。ファジングとは、プログラムにランダムに生成された入力を与えることで、バグや脆弱性を検出する動的解析手法である。ファザーは、文法に基づいてゼロからランダムな入力を生成するか、既存のシードを変更して新たなテストケースを作成する。これらの生成されたテストケースをプログラムに与え、その挙動を監視することで、バグや脆弱性を特定する。また、ファジングの重要なパラメーターの 1 つにカバレッジがある。これは、ファジング中にプログラムがどの程度広範囲にテストされたかを示している。一般的には、ファジング中に少なくとも 1 回実行された制御フローグラフのエッジ数とプログラム内のエッジの合計数の比率として定義される。当然カバレッジが低いと検出対象の見逃しが発生するため、カバレッジを向上させることが多くのファザーにとって重要である。

まず、SpecFuzz がどのように分岐予測による投機実行をシミュレートしているかを説明する。SpecFuzz は、テスト対象のプログラムに対して x86 アーキテクチャ用の LLVM コンパイラバックエンドパスを利用し、投機実行のシミュレーションに必要なコードを計装する。図 2.11 に計装前のプログラムの制御フローグラフと SpecFuzz による計装後の制御フローグラフの概略図を示す。点線が分岐予測ミスによるパス、実線が通常の実行によるパスを表している。また、KLEESpectre と同様に、全ての分岐命令が攻撃者によってトレーニングされていると仮定し、投機実行をシミュレートする。

SpecFuzz では、まず全ての条件分岐命令の直前にチェックポイントを配置する関数 (`checkpoint`) の呼び出しを挿入する。この関数は、投機実行のシミュレーションの開始地点を示し、シミュレーショ

ン終了後に通常の実行パスへ戻るため、現在の CPU 状態を保持する役割を持つ。具体的には、以下の情報をスナップショットとして取得し、メモリに保存する。

- レジスタ値（GPR、フラグ、SIMD、浮動小数点レジスタなど)
- ロールバック先のアドレス
- メタデータ (スタックポインタ、シミュレーション中に実行した命令数など)

次に、全ての分岐命令 (`cond_jump`) を、その分岐条件を反転させた命令 (`!cond_jump`) に置き換える。この操作により、分岐命令に遭遇するたびに通常の実行パスとは逆方向に処理が進み、投機実行のシミュレーションが開始する。さらに、メモリを変更する全ての命令 (mov, push, call など）の直前には、変更対象のアドレスとその直前の値を記録するコードを挿入する。これにより、投機実行のシミュレーション中に行われた全てのメモリの変更がログに記録され、ロールバック時にシミュレーション開始時点の状態に戻すことが可能になる。

投機実行のシミュレーションは、Machine IR（MIR）レベルで実行された命令数が投機ウィンドウによる上限に達するか、直列化命令に遭遇した場合に終了する。各基本ブロックの最後では、シミュレーション中に実行された命令数が投機ウィンドウに達しているかをチェックする関数 (`check_rlbk`) の呼び出しを挿入する。上限に達していない場合、シミュレーションは継続される。一方で、上限に達していた場合、シミュレーションを終了し、直前のチェックポイント地点にシャンプし、スナップショットに基づいてレジスタやメモリの状態を復元した後、正しい実行パスを再開する。

次に、SpecFuzz がどのように Spectre Gadget を検出するかについて説明する。SpecFuzz は、投機実行のシミュレーション中に境界外アクセスが検出された場合、その直前に分岐予測ミスが発生した分岐命令と、境界外アクセスが行われた命令を合わせて Spectre Gadget として報告する。境界外アクセスの検出には、AddressSanitizer（ASan）[35] が使用される。この手法は境界外アクセスが攻撃者の操作している値に依存しているかや、境界外アクセスで読み取った値がキャッシュへ転送されるか (LS が存在するか) といったことは考慮されておらず、単純な検出手法である。そのため、実際には攻撃に利用できない Gadget も多数検出される可能性がある [31]。また、本研究の提案手法とも関連する点として、SpecFuzz はネストされた分岐予測ミスによって悪用可能な Gadget が少ないという観測に基づき、ネストされた分岐予測ミスのシミュレーションを抑制している。そのため、複数回の分岐予測ミスを必要とする Gadget が多数存在した場合、false negative が増加する。しかし、これらの手法によって、他のファジングベースの Spectre Gadget の検出手法 [20, 31] と比較して、より高いファジングスループットを実現している。

SpecFuzz はファジングを活用することで、記号実行を用いた手法と比較して大規模なプログラムに対してもスケールする。しかし、ファジング固有の問題であるカバレッジ不足により、一部の Spectre Gadget を見逃す可能性がある。また、検出機構が単純化されているため、実際には攻撃に利用できない Gadget が誤って検出される可能性もある。

第 3 章

問題設定

本論文では、スケーラビリティと精度を両立する Spectre Gadget の検出手法を提案する。Spectre Gadget を正確かつ効率的に検出することは、最低限の実行時オーバーヘッドでプログラムを Spectre 攻撃に対して堅牢化するために不可欠である。以降では、まず既存の Spectre Gadget の検出手法が抱える問題点について述べる。その後、その問題点を示す Motivating Example を提示し、本研究の目的を明確化する。

3.1 既存手法の問題点

3.1.1 記号実行

記号実行を用いた Spectre Gadget の検出手法 [14, 43] の問題点は、大規模なプログラムに対してスケールしないという点である。これは一般的な記号実行による解析に共通する問題でもあるが、Spectre Gadget の検出ツールは通常のパスに加えて、投機的なパスも考慮する必要があるため、これはより顕著な問題となる。この問題の主な原因はパス爆発とストア状態の爆発として知られている [7]。

パス爆発とは、プログラム内の分岐数に比例して探索すべき実行パスが指数関数的に増加する現象を指す。記号実行エンジンは、分岐箇所に遭遇すると現在の状態を分岐方向ごとに分割し、新たな状態を生成する。このため、分岐が多いプログラムでは実行パスの数が急激に増加し、解析時間やメモリ使用量が膨大となる。結果として、すべての実行パスを網羅的に解析することが現実的に不可能となる場合がある。

パス爆発の主な原因の一つは、ループ構造に起因するものである。ループは分岐命令として扱われるため、各反復ごとに分岐方向に対応する新しい状態が生成される。さらに、図 3.1 に示すように、ループ条件に記号化された変数が含まれる場合、ループの反復回数が具体的に決定できず、最大のループ回数を想定して解析が行われる可能性がある。一般的なツールでは、ループの解析を限られた回数に制限しているが、この場合、検出対象を見逃す可能性もある。

```
1 int i;
2 symbolic_var(i); // Symbolize variable i
3 while(i > 0) {
4   ...
5   i--;
6 }
```

図 3.1: 記号実行におけるパス爆発

```
1 size_t i, j;
2 int temp;
3 symbolic_var(i); // Symbolize variable i
4 symbolic_var(j); // Symbolize variable j
5 int array[5] = {0};
6 if(i < 5 && j < 5) {
7   array[i] = 1;
8   temp = array[j];
9 }
```

図 3.2: 記号実行におけるストア状態の爆発

ストア状態の爆発とは、記号実行において抽象的に扱われるメモリ状態の数が急激に増加する現象を指す。記号実行エンジンは、プログラムのメモリ操作を正確にモデル化するために、各メモリアドレスを具体的な値または記号式に関連付けたストア状態というデータ構造を保持する。そのため、記号化されたアドレスに対して読み取りや書き込みが行われる場合、操作の結果として生じる可能性のある全てのストア状態を考慮する必要がある。その結果、新しいストア状態が次々に生成され、ストア状態の数が急激に増加する。この現象は、大規模なプログラムや複雑なメモリアクセスパターンを持つコードにおいて特に顕著である。

図 3.2 に示すコード例を用いて、ストア状態の爆発について具体的に説明する。このコードでは、変数 `i` と `j` が記号化されている。まず、7 行目の配列への書き込みに注目する。この箇所では、`i` が記号化されているため、`array[0]` から `array[4]` のいずれかの要素が 1 に書き換えられる可能性がある。その結果、これら 5 つの可能性を全て考慮するため、7 行目の実行後に現在の状態から 5 つの新しい状態が分岐される。同様に、8 行目の配列への読み取りにおいても、`j` が記号化されているため、`array[0]` から `array[4]` のいずれかの要素が読み取られ、変数 `temp` に格納される。この場合も、5 つの可能性を考慮するため、8 行目の実行後にさらに 5 つの新しい状態が分岐し、最終的には最初の状態から新しく 25 個の状態に分岐することになる。

この例では、変数 `i,j` の範囲が制限されていたため、メモリ操作が参照する可能性のあるアドレスのセットは比較的小さかったが、一般的には、記号化されたアドレスはメモリ内の任意のアドレスを参照する可能性があるため、ストア状態の数が爆発的に増加する可能性がある。

パス爆発とストア状態の爆発の問題は、記号実行を用いて Spectre Gadget を検出する場合により顕著になる。これは、分岐予測ミスによる投機的なパスを考慮することで、通常の記号実行に比べて探索すべきパス数が大幅に増加するからである。通常の実行パス上の状態と異なり、投機的なパス上の状態は投機ウィンドウの上限に達した時点で破棄されるため最後まで残ることはない。しかし、それでも解析時間や最大メモリ使用量に大きな影響を与えるため、大規模なプログラムや複雑な制御フローを持つプログラムに対してはスケールしない。

3.1.2 ファジング

ファジングを用いた Spectre Gadget の検出手法における主な問題点は、カバレッジ不足により Gadget の見逃しが発生する可能性がある点である。ファジングは記号実行とは異なり、生成された入力によって実行されたパス上に存在する Gadget しか検出できない。これは一般的なファジングツールに共通する問題でもある。

さらに、ファザーが脆弱性を引き起こす入力を生成しない場合、Spectre Gadget を検出することは

できない。例えば、図 2.4 に示すコードを考える。ここで、ファザーが条件 `i < array1_size` を満たす入力 `i` を生成した場合、3 行目および 4 行目は通常実行される。そのため当然、3 行目で境界外アクセスが発生しないので、脆弱性は検出されない。一方、`i < array1_size` を満たさない入力 `i` を生成することで、3 行目及び 4 行目が投機実行がシミュレートされ、3 行目の境界外アクセスが引き起こされ、初めて脆弱性を検出できる。このように、ファジングによる Spectre Gadget の検出は、通常の実行パスだけでなく、投機的なパスも漏れなくテストすることが必要である。

3.2 Motivating Example

図 3.3 は、記号実行を用いた Spectre Gadget の検出において、投機的な状態数が爆発する具体例を示している。この図には、3 つの Spectre Gadget が含まれている。1 つ目は、18 行目が分岐予測ミスされた後に実行される 9 行目の命令である。2 つ目は、9 行目と 18 行目の両方が分岐予測ミスされた後に実行される 10 行目の命令である。そして 3 つ目は、18 行目、9 行目、11 行目の全てが分岐予測ミスされた後に実行される 12 行目の命令である。先述の通り、CPU はネストされた投機実行が可能であり、既存の記号実行による手法 [14, 43] では、これらの Gadget を漏れなく検出するため、ネストされた分岐予測ミスをシミュレートする必要がある。そのため、投機ウィンドウの上限に達するまで、分岐命令に遭遇するたびに以下の 4 つの新たな状態が生成され、状態数の爆発に大きく寄与していると考えられる。

- 分岐条件を満たし、Taken 側へ遷移した状態
- 分岐条件を満たさず、Not taken 側へ遷移した状態
- 分岐条件を満たさず、分岐予測ミスにより Taken 側へ遷移した状態
- 分岐条件を満たし、分岐予測ミスにより Not taken 側へ遷移した状態

しかし、既存研究 [30] では、現実世界の検体において、10 行目や 12 行目のような、ネストされた分岐予測ミスによりトリガーされる Gadget は少ないことが報告されている。この直感的な理由として、多くのメモリアクセス命令は、単一の境界チェック条件で保護されており、10 行目や 12 行目のような複数の境界チェックで保護されているメモリアクセス命令は稀であるからと考えられる。また、複数の分岐命令を訓練することは、攻撃者にとって非常に困難であるため、このような Gadget が実際の攻撃で利用される可能性は低いと考えられる。

既存の記号実行による手法では、Gadget が検出される可能性の低い (あるいは、検出しても攻撃に利用されにくい) ネストされた分岐予測ミスにより到達する投機的な状態まで探索しているため、スケーラビリティが低下していると考える。

表 3.1 に図 3.3 を KLEESpectre で解析を行った結果を示す。実験環境及び構成は 6.3 節の通りである。表 3.1 からわかる通り、小規模な検体でありながら、通常実行において探索した状態数 (State) に比べて、非常に多くの投機的な状態 (Speculative states) を探索している。これにより、通常の記号実行による解析と比較して解析時間と最大メモリ使用量が大幅に増加し、スケーラビリティが低下していると考える。

表 3.1: KLEESpectre による Motivating Example の解析結果

Detected gadgets	**States**	**Speculative states**	**Analysis Time (h:m:s)**	**Max Memory Usage (KB)**
3	17	38039	0:0:21	66968

```
#define ARRAY1_SIZE 16
uint8_t array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
uint8_t array2[256 * 512];
uint8_t temp = 0;

void var(size_t index) {
    if (index < ARRAY1_SIZE / 2) {   // b2
        // C
        temp &= array2[array1[index] * 512];
        if (index < ARRAY1_SIZE / 4) {  // b3
            // D
            temp &= array2[array1[index] * 512];
        }
    }
    // E
}

void foo(size_t index) {
  while (index < ARRAY1_SIZE) {  // b1
    // A
    temp &= array2[array1[index] * 512];
    var(index);
    ++index;
  }
    // B
}

int main() {
  size_t num = input();
  foo(num);
  return 0;
}
```

図 3.3: Motivating Example

第 4 章

提案手法

本研究では、記号実行とファジングの解析手法を組み合わせることで、スケーラビリティと精度の両立を目指す Spectre Gadget 検出手法を提案する。記号実行とファジングにはそれぞれ利点と欠点が存在するが、これらを組み合わせることで、両者の欠点を補完し、スケーラビリティと精度を両立できるのではないかと考える。提案手法は、記号実行フェーズとファジングフェーズの 2 つのフェーズで構成され、それぞれのフェーズで検出された Gadget を合わせて、最終的な検出結果として報告する。記号実行フェーズでは、ネストされた分岐予測ミスの回数を制限することで、投機的状態の探索範囲を縮小し、記号実行のスケーラビリティを向上させる。ファジングフェーズでは、記号実行フェーズで探索されなかった投機的な状態を集中的にテストすることで、記号実行で見逃された Gadget を効率的に検出することを目指す。

以下では、記号実行フェーズとファジングフェーズのそれぞれについて、提案手法の詳細を説明する。

4.1 記号実行フェーズ

4.1.1 ネストされた分岐予測の制限

記号実行フェーズでは、ネストされた分岐予測ミスの回数を制限することでスケーラビリティの向上を図る。提案手法において、ネストされた分岐予測ミスのシミュレーションの最大回数を Order と定義する [30]。例えば、Order1 の解析では、シミュレーションは 1 回のみ行われ、それ以降、その状態では分岐命令に遭遇してもシミュレーションを行わない。記号実行フェーズでは、この Order 値をユーザが指定することで、ネストされた分岐予測ミスを制限する。

Order1 の記号実行フェーズがどのように分岐予測ミスをシミュレートするかについて、図 3.3 の制御フローグラフの一部である図 4.1 を用いて説明する。点線が分岐予測ミス、実線が通常の実行パスを表す。

まず、既存手法と同様に、解析が分岐 `b1` に到達すると、シミュレーションが開始され、以下の 4 つの状態に分岐する。

1. 分岐 `b1` の条件を満たし、A に遷移した状態
2. 分岐 `b1` の条件を満たさず、B に遷移した状態
3. 分岐 `b1` の条件を満たさず、分岐予測ミスにより A に遷移した状態
4. 分岐 `b1` の条件を満たし、分岐予測ミスにより B に遷移した状態

これらのうち、3 番目の状態のみが 23 行目の脆弱性を引き起こし、Spectre Gadget として検出される。この状態から解析を進めていくと、次に分岐 `b2` に遭遇する。既存手法では、ネストされた分岐予測ミスをシミュレートするため、`b2` でも分岐予測ミスを含めた 4 つの状態に分岐する。一方、提案手

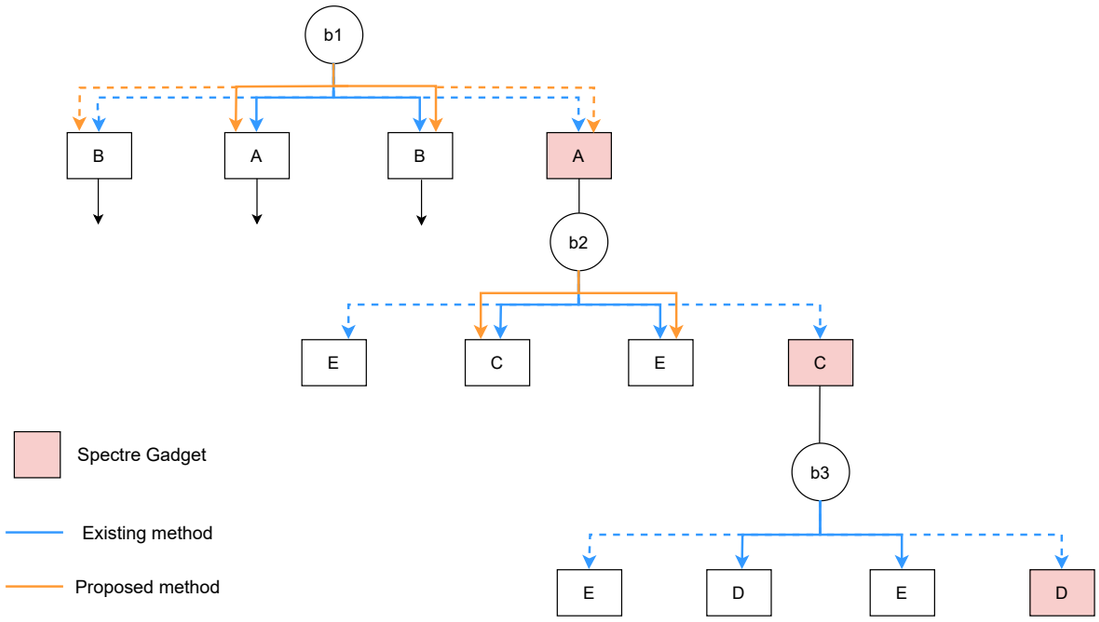

図 4.1: Order1 の記号実行フェーズの解析

法ではネストされた分岐予測ミスが制限されているため、b2 における分岐予測ミスによる投機的な状態は探索しない。そのため、この状態において、以降は、投機ウィンドウの上限に達するか、直列化命令に遭遇するまで、正しい分岐方向の状態のみを探索し続ける。

その結果、11 行目と 14 行目の脆弱性を引き起こす Gadget が見逃されてしまう。しかし、3.2 で述べた通り、ネストされた分岐予測ミスにより引き起こされる Gadget は稀であるため、影響は少ないと考える。また、このようなネストされた分岐予測ミスにより到達する投機的な状態は、後述するファジングフェーズで探索を行うことで、false nagative を最小限に抑え、精度を担保できると考える。

4.1.2 不要な分岐予測ミスの特定

記号実行で探索されなかったネストされた分岐予測ミスにより到達する投機的な状態はファジングフェーズで探索を行う。以降は、このような状態をターゲット状態と呼ぶ。この際、ファザーでターゲット状態に到達しない分岐予測ミスをシミュレートすることは非効率的である。

図 4.2 に示すコード例を用いて、具体的に説明する。変数 `index` は攻撃者によって操作可能であり、ファザーによって与えられる入力値を保持すると仮定する。このコード例をファジングフェーズで解析すると、`index < ARRAY1_SIZE` を満たさない入力が `index` に与えられた場合、5 行目に到達すると分岐予測ミスのシミュレーションが開始し、6 行目に実行が進む。その結果、6 行目の脆弱性が引き起こされる。しかし、これは 1 回の分岐予測ミスによって引き起こされる脆弱性であり、Order1 の記号実行フェーズですでに検出済みである。その後、シミュレーションは続行されるものの、7 行目以降に投機ウィンドウ以上の命令数や直列化命令が存在する場合、8 行目に到達する前にシミュレーションが終了し、5 行目にロールバックされて正しいパスが実行されることになる。

このように、ファジングフェーズにおいて、特定の分岐方向へ分岐予測ミスをシミュレートしても、ターゲット状態に到達できない場合がある。このようなシミュレーションを行っても、記号実行フェーズで既に探索された投機的な状態を再度探索するだけであり、ファジングのスループットを低下させる原因となる。以降ではこのような分岐予測ミスのシミュレーションが不要な分岐方向を Removable Direction と呼ぶ。記号実行の過程では、この Removable Direction を特定し、その情報を元にファジ

```
uint8_t array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
uint8_t array2[256 * 512];
uint8_t temp = 0;
void var(size_t index) {
    if (index < ARRAY1_SIZE) {
       temp &= array2[array1[index] * 512]; // Discovered by symbolic execution
      // lots of code or lfence
      if (condition) {
        // more code
      }
    }
}
```

図 4.2: ファジングフェーズにおける分岐予測ミスのシミュレートが不要な例

ングフェーズでは、Removable Direction の分岐予測ミスのシミュレートを抑制する。

Algorithm1 に記号実行の過程で Removable Direction を特定するプロセスを示す。各条件分岐命令から分岐予測ミスのシミュレーションが開始された場合、シミュレーションが終了するまでに Order 値 (`Order`) 以上の分岐命令に遭遇するかを分岐方向ごとに記録する。記号実行における各状態 (μ) は、シミュレーションを開始した分岐命令 (`missBranch`) と分岐予測ミスにより進んだ分岐方向 (`missDirection`) の情報を保持している。また、その分岐方向への分岐予測ミスのシミュレートが不要かどうかを表すフラグ `isRemovable` を保持しており、`true` に初期化されている。

シミュレーション中に Order 値の制限に達した場合、その分岐方向へ分岐予測ミスをシミュレートすることでターゲット状態に到達することを意味するため、`isRemovable` を `false` にする。最終的に、シミュレーションが終了した際、分岐命令の情報 (`missBranch, missDirection`) と `isRemovable` の値を記録する。ただし、同一の分岐命令の分岐方向において、複数の結果が得られた場合 (つまり、`isRemovable` が `true` の場合と `false` の場合が存在した場合)、その分岐方向への分岐予測ミスは必要であると保守的に扱う。

最終的な結果は JSON 形式で図 4.3 のように出力される。`location` は条件分岐命令の位置を示し、`truePath` はその分岐命令の Taken 側への分岐予測ミスのシミュレートが不要な場合に `true` となる。同様に、`falsePath` は Not taken 側への分岐予測ミスのシミュレートが必要かを示している。この情報を元に、ファジングフェーズでは分岐予測ミスのシミュレートを抑制する。

4.1.3 ターゲット状態までの実行トレースの取得

ファジングフェーズでは、ターゲット状態に到達する入力を効率的に生成することが求められる。そのため、テストケースがターゲット状態に到達する可能性を評価するための指標が必要となる。その指標として、記号実行フェーズにおいて、ターゲット状態に至るまでに通過した分岐方向 (分岐予測ミスも含む) の実行トレースを収集する。以降ではこのような分岐方向を Target Direction と呼ぶ。この情報を基に、ファジングフェーズではテストケースが Target Direction をどれだけ通過できたかで、そのテストケースの有効性を評価する。

記号実行では、各状態は、通過した分岐命令と進んだ分岐方向の情報を保持している。Order の制限に達した時点で、その状態が保持する分岐情報のトレースをグローバルなデータ構造に記録する。最終的に、各分岐命令の分岐方向ごとに、ターゲット状態に到達するまでにその分岐方向を一度でも通過し

Algorithm 1 Removable Direction を特定するアルゴリズム

```
Input:
    μ_0: Initial state
    φ_r: Branch condition for instruction r
    SEW: Speculative execution window
    Order: Maximum nested branch misprediction limit
function SYMBOLIC_EXECUTION(μ_0, SEW, Order)
    states ← {μ_0}
    while states is not empty do
        μ ← selectState(states)
        if μ is speculative state and μ.instructionCount > SEW then
            recordMissBranch(μ.isRemovable, μ.missBranch, μ.missDirection)
            terminateSpState(μ)
            continue
        end if
        r ← μ.nextInstruction()
        if μ is speculative state then
            μ.instructionCount ← μ.instructionCount + 1
            if r is a conditional branch then
                if μ.missBranchCount < Order then
                    μ_t ← createSpState(μ, ¬φ_r)
                    μ_f ← createSpState(μ, φ_r)
                    μ_t.missBranchCount ← μ_t.missBranchCount + 1
                    μ_f.missBranchCount ← μ_f.missBranchCount + 1
                    if μ.missBranchCount = 1 then
                        μ_t.missBranch ← r
                        μ_t.missDirection ← true
                        μ_t.isRemovable ← false
                        μ_f.missBranch ← r
                        μ_f.missDirection ← false
                        μ_f.isRemovable ← false
                    end if
                    states ← states ∪ {μ_t, μ_f}
                else
                    μ.isRemovable ← false
                end if
            end if
            if r is Exit or Fence then
                recordMissBranch(μ.isRemovable, μ.missBranch, μ.missDirection)
                terminateSpState(μ)
            end if
        end if
    end while
end function
```

```
{
  "branches": [
    {
      "location": "/path/sample.c:12",
      "truePath": false,
      "falsePath": true,
    },
    {
      "location": "/path/sample.c:18",
      "truePath": true,
      "falsePath": false,
    }
    ...
  ]
}
```

図 4.3: Removable Direction を特定した結果

```
{
    "branches": [
        {
            "location": "/path/sample.c:9",
            "truePath": 1,
            "falsePath": 0,
            "trueSpPath": 0,
            "falseSpPath": 0
        },
        {
            "location": "/path/sample.c:21",
            "truePath": 1,
            "falsePath": 1,
            "trueSpPath": 0,
            "falseSpPath": 1
        },
       ...
    ],
    "Sum": 14
}
```

図 4.4: Target Direction を取得した結果

たかどうかを集計し、Target Direction を JSON 形式でファイルに出力する。

図 4.4 に出力される JSON ファイルの内容を示す。`location` は条件分岐命令の位置を示し、その後分岐予測ミスを含んだ各分岐方向 (`truePath`,`falsePath`,`trueSpPath`,`falseSpPath`) に対し、ターゲット状態に到達するまでにその分岐方向を一度でも通過した場合は 1 を、通過しなかった場合は 0 が記録される。`Sum` はターゲット状態に到達するまでに通過した分岐方向 (すなわち 1 が記録されている分岐方向) の総数を表す。この情報を元に、ファジングフェーズではテストケースの有効性を評価する。

4.2 ファジングフェーズ

4.2.1 不要な分岐予測ミスの抑制

ファジングフェーズでは、記号実行で探索されなかったターゲット状態を集中的に探索したい。その際、ターゲット状態に到達しない分岐予測ミスをシミュレートすることは非効率的である。そこで、記

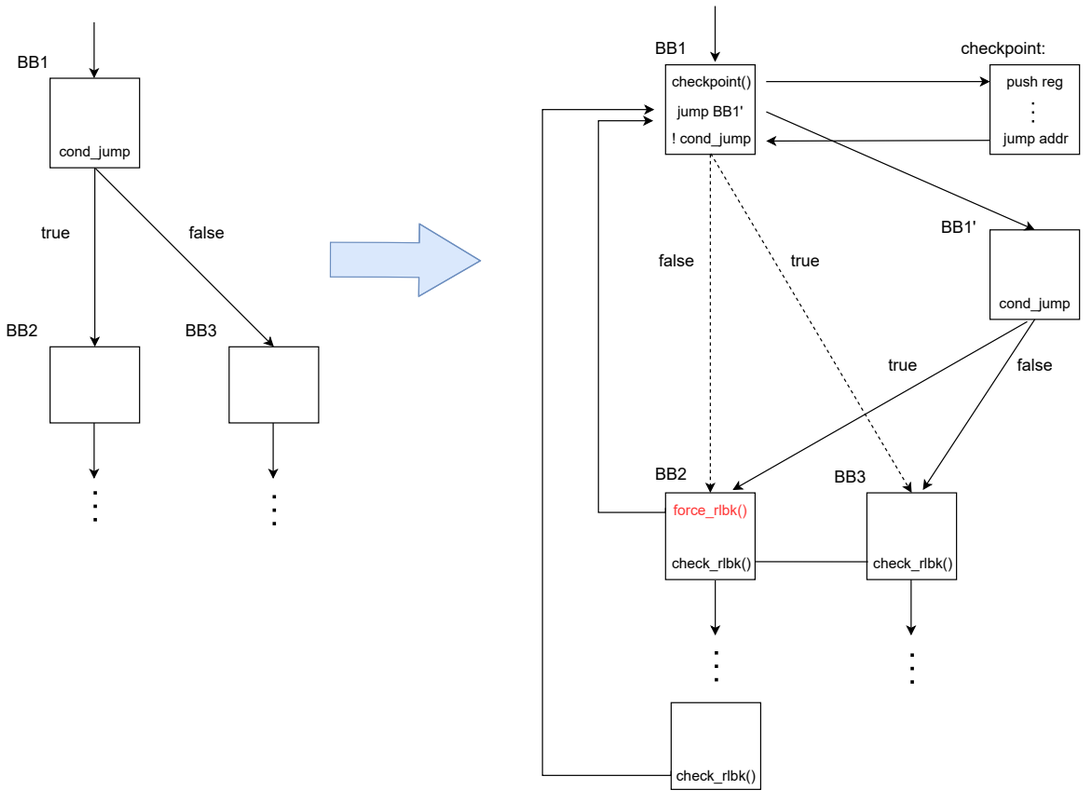

図 4.5: 計装による Removable Direction の分岐予測ミスの抑制

号実行フェーズで得られた結果 4.3 を用い、Removable Direction の分岐予測ミスのシミュレートを抑制する。

提案手法は LLVM の MIR レベルでの計装によって行われる。図 4.5 に計装前の制御フローグラフと、分岐予測ミスを抑制した計装後の制御フローグラフの概略図を示す。分岐予測ミスのシミュレート用の計装は SpecFuzz と同様である。詳しくは 2.5.2 を参照されたい。提案手法では、記号実行フェーズの結果から、Removable Direction であると判断された分岐方向に対応する基本ブロックの先頭に、`force_rlbk()` という関数の呼び出しを挿入する。この関数は、直前の分岐命令においてシミュレーションが開始されていた場合に、シミュレートを終了し、強制的にロールバックさせる関数である。このような計装を行うことで、投機ウィンドウの上限に達するのを待たずにシミュレーションを終了できるため、記号実行で既に探索済みの投機的な状態を再度探索することをできるだけ回避する。その結果、ファジングフェーズのスループット向上が期待される。

4.2.2 スコアによるシードのスケジューリング

ファジングフェーズでは、ターゲット状態に到達する入力を効率的に生成することが求められる。そこで、記号実行フェーズで得られた結果 4.4 を用いて、テストケースが Target Direction をどれだけ通過したかでテストケースの有効性を評価する。ファジングフェーズでは、LLVM の MIR レベルでの計装によって、テストケースが通過した分岐命令と分岐方向 (分岐予測ミスも含む) を記録するようにする。そして、式 4.1 により導出した値の小数点以下を四捨五入した値をテストケースのスコアとする。

$$\text{スコア} = \left(\frac{\text{テストケースが通過した Target Direction の種類数}}{\text{Target Direction の総数}} \right) \times 10 \tag{4.1}$$

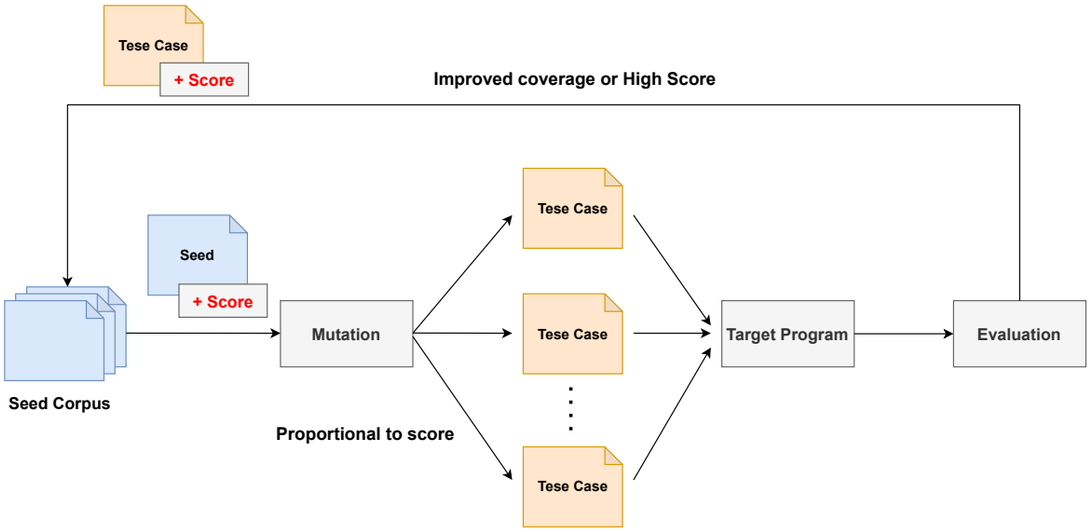

図 4.6: スコアを用いたシードのスケジューリング戦略

通過した分岐方向の種類数をスコア計算に使用するため、ループなどの影響で同一の分岐方向に複数回通過しても、一度しかカウントしない。これにより、テストケースがループなどによって不適切に高いスコアを獲得することを防ぐ。このスコアをファザーへのフィードバックとして提供する。ファザーは高いカバレッジを獲得したテストケースに加え、より高いスコアを獲得したテストケースもシードコーパスに追加する。また、シードが保持するスコアが高いほど、そのシードを元に変異によって生成される新しいテストケース数を増やすようにする。図 4.6 にスコアを用いたシードのスケジューリング戦略の概要を示す。こうすることで、ファザーはターゲット状態に到達する可能性の高いテストケースを優先的に実行できると考える。

第 5 章

実装

提案手法の全体像を図 5.1 に示す。提案手法では、記号実行を KLEESpectre を元に、ファザーを SpecFuzz を元に実装した。記号実行フェーズでは、LLVM IR を対象に Order による制限を加えた記号実行を行い、解析結果をファジングフェーズに提供する。ファジングフェーズでは、LLVM MIR レベルの Pass を用いて、分岐予測ミスのシミュレーション用の計装と、記号実行フェーズで得られた解析結果に基づく計装を施したバイナリを生成する。この計装済みのバイナリを、カバレッジ駆動型のファジングツールである honggfuzz [1] を使用してファジングを行う。スコアに基づくシードのスケジューリングは、honggfuzz を拡張することで実現している。以降では、提案手法の詳細な実装について詳しく説明する。

5.1 記号実行フェーズ

5.1.1 Order による分岐予測ミスの制限

Order 値は、ユーザがコマンドラインオプションを通じて設定可能であり、デフォルト値は 1 に設定されている。記号実行における状態を表現するデータ構造に、現在のネストされた分岐予測ミスの回数を追跡するためのメンバ変数 `specBranchCount` を追加し、新たな投機的状態がフォークされるたびに、`specBranchCount` をインクリメントする。また、投機的な状態をフォークする前に、現在の状態における `specBranchCount` が設定された Order 値を超えていないかを確認する。Order 値を超えていない場合は通常通りフォーク処理が行われるが、超えている場合はフォーク処理を行わず、投機的な状態の生成を抑制する。

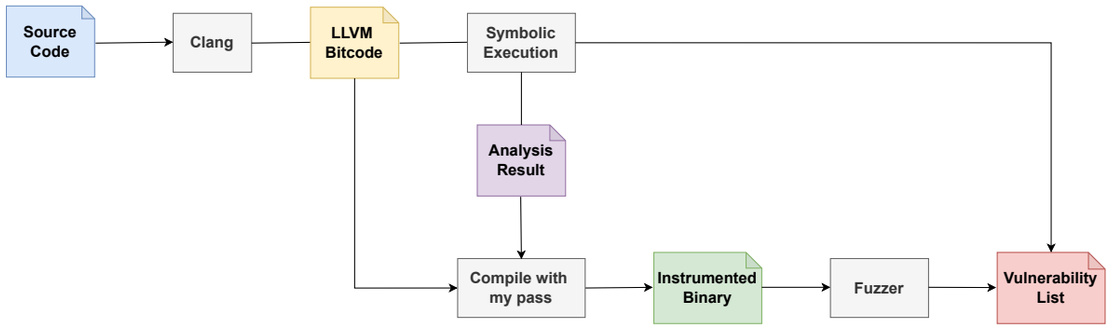

図 5.1: 提案手法の全体像

5.1.2 異常終了した投機的な状態の扱い

記号実行では、いくつかの要因により投機的な状態が正常に終了しない場合がある。たとえば、ユーザが指定したメモリ使用量の上限やスタック数の上限に達した場合、一部の状態の探索が途中で打ち切られることがある。記号実行フェーズで Removable Direction を特定する際、このような理由で探索が途中で終了し、ターゲット状態に到達できなかった場合、解析が継続していれば到達可能だったかどうかを判断することができない。そのため、このようなケースではその分岐予測ミスによりターゲット状態に到達する可能性があると保守的に判断し、Removable Direction ではないと記録する。逆に Removable Direction として記録される可能性があるのは、投機的な状態が正常に終了した場合のみである。正常に終了したと判断されるのは以下のいずれかの場合のみである。

- 投機ウィンドウの上限に達した場合
- 直列化命令に遭遇した場合
- プログラムが終了した場合

5.2 ファジングフェーズ

5.2.1 分岐予測ミスの抑制を行う計装

Removable Direction の分岐予測ミスのシミュレーションを抑制するための計装を行うアルゴリズムを Algorithm2 に示す。まず各基本ブロックの終端命令を確認し、それが分岐命令であるかを調べる。分岐命令である場合、記号実行の解析結果 4.3 内に該当する分岐命令が存在するかを確認する。この確認は、分岐命令の位置情報 (`location`) が一致するかどうかで行う。Removable Direction であると判断されている場合 (`truePath` または `falsePath` が `true` である場合)、該当分岐方向の遷移先となる基本ブロックの先頭に、ランタイム関数 `force_rlbk` の呼び出しを挿入する。ランタイム関数 `force_rlbk` は、直前の分岐命令が分岐予測ミスされていた場合、シミュレーションを終了させ、即座にロールバックさせる。この計装により、Removable Direction の分岐予測ミスが発生した場合でも、即座にシミュレーションを終了させることができる。

5.2.2 スコア計算のための計装

スコア計算を行うために、テストケースがどの分岐方向を通過したかを記録するための計装を行う。まず、Algorithm2 と同様に、対象プログラム内の分岐命令を特定し、記号実行の解析結果 4.4 内に該当する分岐命令が存在するかを確認する。該当する分岐命令における各分岐方向が Target Direction であると判断されている場合は (`truePath`,`falsePath`,`trueSpPath`,`falseSpPath` のいずれかが 1 の場合)、その分岐方向を一意に識別するためのインデックスを割り当てる。この時、投機的な分岐方向にも一意のインデックスが割り当てられる。そして、インデックスが割り当てられた分岐方向の遷移先となる基本ブロックの先頭に、ランタイム関数 `record_direction` の呼び出しを挿入する。このランタイム関数は分岐方向を識別するインデックスを引数として受け取り、そのインデックスに対応するビットマップ上の位置にビットを立てる。つまり、テストケースが通過した Target Direction の種類をビットマップを用いて記録する。最終的に、テストケースが通過した Target Direction の種類数をビットマップから取得し、式 4.1 に基づきスコアを計算する。

Algorithm 2 Removable Direction の分岐予測ミスの抑制のための計装を行うアルゴリズム

```
Input:
    CFG: Control Flow Graph of the target program at LLVM MIR level
    Result: Analysis results containing unnecessary branch misprediction information
function SUPPRESS_MISPREDICTION(Result, CFG)
    for all BasicBlock B in CFG do
        Terminator ← B.getTerminator()
        if Terminator.isConditionalBranch() then
            Location ← Terminator.getLocation()
            BranchInfo ← findBranchInfo(Result, Location)
            if BranchInfo exists then
                if BranchInfo.truePath = true then
                    TrueBlock ← Terminator.getTakenSuccessor()
                    addCallRuntimeFunction(TrueBlock, "force_rlbk")
                end if
                if BranchInfo.falsePath = true then
                    FalseBlock ← Terminator.getNotTakenSuccessor()
                    addCallRuntimeFunction(FalseBlock, "force_rlbk")
                end if
            end if
        end if
    end for
end function
```

5.2.3 投機ウィンドウの設定

KLEESpectre と SpecFuzz の違いの一つに、投機実行のシミュレーション中における命令数のカウントの基準が挙げられる。KLEESpectre は解析対象が LLVM IR であるため、LLVM IR レベルの命令数をカウントし、投機ウィンドウの上限に達しているかをチェックする。一方で、SpecFuzz は LLVM MIR レベルの命令数をカウントするコードを計装し、実行時に投機ウィンドウの上限を確認する。この違いにより、単純に 2 つのツールを併用して同じ投機ウィンドウの上限を設定しても、カウントされる命令レベルが異なるため、投機実行のシミュレーション終了のタイミングにずれが生じ、検出される Gadget に差が生じる可能性がある。そこで、SpecFuzz の命令数のカウントを LLVM IR レベルで行うように修正し、KLEESpectre と基準を統一した。

LLVM IR レベルのパスを用いて、解析対象のコードに実行時の LLVM IR レベルでの命令数をカウントするコードを計装する。シミュレーション中に実行された命令数はグローバル変数 `instruction_counter` で管理される。具体的には、各基本ブロックの先頭に、その基本ブロックに含まれるデバッグ命令を除いた命令数を `instruction_counter` に加算するコードを挿入する。シミュレーションが開始されると、`instruction_counter` は 0 に初期化され、その後、各基本ブロックの先頭で命令数がインクリメントされる。そして、`instruction_counter` の値が投機ウィンドウの上限に達しているかは、各基本ブロックの終端でチェックされる。このチェックに基づいて、シミュレーションを継続するか否かが判断される。

第 6 章

予備実験

本章では、提案手法の評価を目的として、以下の Research Question（RQ）に答えるべく予備実験を行う。

- (RQ1) 記号実行フェーズにおける Order 制限はスケーラビリティを向上させるか
- (RQ2) ファジングフェーズにおける不要な分岐予測ミスの抑制とスコアによるシードのスケジューリング戦略は効果的か
- (RQ3) 解析全体を通して、既存手法 [30, 43] よりも効率的に Gadget を検出できるか

以降では、実験における比較対象、データセット、および環境について概説した後、提案手法の有効性を評価する。

6.1 比較対象

提案手法は、KLEESpectre [43] および SpecFuzz [30] と比較を行う。KLEESpectre は記号実行を拡張し、投機的なパスを探索することで Spectre Gadget を検出するツールである。一方、SpecFuzz はファジングを拡張し、投機的実行のシミュレーション中に発生する境界外アクセスを検出することで Gadget を検出するツールである。これらはいずれも、Spectre Gadget の静的解析および動的解析手法の代表的なツールであり、提案手法の実装のベースともなっているため、比較対象とした。なお、SpecTaint [31] は SpecFuzz と同様にファジングを用いる手法であり、テイント解析を活用することで SpecFuzz より少ない誤検出で Gadget を検出する。しかし、実装が公開されていないため、本研究では比較対象から除外した。

6.2 データセット

提案手法の Spectre Gadget の検出能力を評価するにあたり、実世界のプログラムにおける ground truth(GT) が限られているため、定量的な比較が難しい。この問題に対処するため、既知の Spectre Gadget を実世界のプログラムに埋め込むことで、評価用のデータセットを作成した。埋め込む Gadget は、Paul Kocher が作成した Spectre V1 のサンプルコード [21] から選択した。提案手法の有効性を評価するために、選択した Gadget をベースに以下の 3 種類の Gadget を作成して埋め込んだ。

- Type1: 1 回の分岐予測ミスで悪用可能な Gadget。
- Type2: 2 回のネストされた分岐予測ミスで悪用可能な Gadget。
- Type3: 3 回のネストされた分岐予測ミスで悪用可能な Gadget。

以降では、この 3 種類の Gadget をそれぞれ Type1, Type2, Type3 と表記する。実際に埋め込んだ

```
spec_idx = input(); // controllable via input

// Type1
if (spec_idx < ARRAY1_SIZE) {
  temp &= array2[array1[spec_idx] * 512];
}
// Type2
if (spec_idx < ARRAY1_SIZE) {
  if (spec_idx < ARRAY1_SIZE) {
    temp &= array2[array1[spec_idx] * 512];
  }
}
// Type3
if (spec_idx < ARRAY1_SIZE) {
  if (spec_idx < ARRAY1_SIZE) {
    if (spec_idx < ARRAY1_SIZE) {
      temp &= array2[array1[spec_idx] * 512];
    }
  }
}
```

図 6.1: 対象プログラムに埋め込んだ 3 種類の Spectre Gadget

Gadget を図 6.1 に示す。

実世界のプログラムには、広く利用されている暗号化ライブラリである OpenSSL [2] v3.3.0 から 7 つの暗号化関連プログラムを選択した。これらのプログラム内で制御フローに影響を与えない箇所に Gadget を埋め込み、それらを入力から制御できるようにするコードを追加した。そして、合計 20 個の Gadget を 7 つのプログラムに対して埋め込んだ。各検体ごとに埋め込む Gadget 数は検体のサイズに応じて決定し、1 回の分岐予測ミスで悪用可能な Gadget（Type1）と、複数のネストされた分岐予測ミスで悪用可能な Gadget(Type2 および Type3) のそれぞれが少なくとも各プログラムに 1 つ以上含まれるようにした。作成したデータセットの概要を表 6.1 に示す。以降では、ここで埋め込んだ Gadget を ground truth(GT) として評価を行う。

6.3 実験環境及び構成

実験は以下の環境で行った。

- OS: Ubuntu 18.04.6 LTS
- CPU: Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz
- RAM: 128GB
- Clang: 7.0.1(SpecFuzz), 6.0.0(KLEESpectre、提案手法)

SpecFuzz と提案手法および KLEESpectre では、必要とする Clang のバージョンが異なるため、それぞれ対応するコンパイラを使用した。また、LLVM pass での計装がコンパイル時の最適化により無効化される可能性を排除するため、全ての検体のビルド時に最適化オプション -O0 を指定した。 提案

表 6.1: データセットの概要

Program	GT (Type1/Type2/Type3)	Loc	Branch
AES (Advanced Encryption Standard)	2/1/0	1166	25
ARIA	2/1/1	736	18
CAST (Carlisle Adams and Stafford Tavares)	1/0/1	302	16
DES (Data Encryption Standard)	2/2/0	805	21
IDEA (International Data Encryption Algorithm)	1/1/0	216	11
MD2 (Message Digest Algorithm 2)	1/1/0	230	11
RC5 (Rivest Cipher 5)	1/1/1	294	21

GT(Type1/Type2/Type3): 埋め込んだ各 Type の Gadget 数. Loc: テストドライバを含んだ検体の行数. Branch: 条件分岐命令の数

手法の Order 値は 1 に設定して実験を行った。つまり、記号実行フェーズでは Type1 の Gadget を検出し、ファジングフェーズでは Type2 及び Type3 の Gadget を検出することが目的となる。また、投機ウィンドウのサイズは 200 に設定した。ここで、SpecFuzz は LLVM MIR レベルで投機ウィンドウを計測するのに対し、提案手法および KLEESpectre では LLVM IR レベルで投機ウィンドウを計測することに注意されたい。埋め込んだ Gadget はすべて、分岐予測ミスから境界外アクセスまでの命令数が少ないため、この違いによって特定の Gadget が一方のツールだけで見逃されることはない。各検体に対するテストドライバはそれぞれ自作したものを使用した。

6.4 実験結果

6.4.1 記号実行フェーズの評価

RQ1 に回答するため、提案手法の記号実行フェーズを評価した。具体的には、提案手法の記号実行フェーズと KLEESpectre を用いてデータセットを解析し、探索された投機的状態数、解析時間、メモリ使用量を比較することで、Order 制限が記号実行のスケーラビリティ向上に有効であるかを検証した。各プログラムの入力を記号変数として設定し、KLEE のオプションを使用して最大メモリ使用量を 8GB に制限して実験を行った。表 6.2 と表 6.3 に、提案手法の記号実行フェーズと KLEESpectre によるデータセットの解析結果を示す。表 6.4 では、KLEESpectre に対する提案手法の記号実行フェーズにおける投機的状態数、解析時間、メモリ使用量の割合を示している。

表 6.2,6.3 に示す通り、Order 制限によりネストされた分岐予測ミスが 1 回に制限されているため、提案手法の記号実行フェーズでは Type1 の Gadget のみが検出されていることがわかる。また、KLEESpectre および提案手法のいずれもいくつかの Gadget を見逃している。この原因として、記号化する変数に漏れがあった可能性や、解析時に指定した KLEE のオプションの影響が考えられる。さらに、KLEESpectre で IDEA を解析した際、最大メモリ使用量の上限である 8GB に達したため、状態のフォークが抑制されたことが確認された。これにより、通常の状態数が提案手法の記号実行フェーズの結果と比較して大幅に減少している。

表 6.4 に示す通り、全ての検体で Order 制限により解析時間が削減されたことが確認された。特に、IDEA、MD2、RC5 においては、KLEESpectre と比較してそれぞれ解析時間が 1 時間以上短縮された。また、最大メモリ使用量についても、AES を除く全ての検体で削減が見られた。特に、IDEA、MD2、RC5 では最大メモリ使用量が約 70% 以上削減された。これらの差異は、Order 制限による投機的状態数の削減と関連していると考えられる。IDEA、MD2、RC5 では、KLEESpectre と比較して探索され

た投機的状態数が約 1% にまで抑制された。一方、他の検体では投機的状態数が十数 % 程度までしか抑制されておらず、この違いが解析時間および最大メモリ使用量の削減度合いに影響を与えたと考える。KLEESpectre では、投機的状態が投機ウィンドウに達した時点で削除される最適化が行われている。その結果、通常の状態とは異なり、解析が終了するまで記号実行エンジンによって投機的状態が保持されることはない。このため、AES、ARIA、CAST、DES などの検体では他の検体と比較して、投機的状態数が大幅に削減されなかったため、解析時間及び最大メモリ使用量に大きな影響を与えなかったと考えられる。以上から、Order 制限により投機的状態数が大幅に削減される場合、記号実行のスケーラビリティが大きく向上すると思われる。特に、ループ構造を多く含む検体では投機的状態数が増加しやすく、そのような検体に対しては高い効果を発揮すると考えられる。

表 6.7 は、全ての分岐方向に対する提案手法の記号実行フェーズで特定した Removable Branch と Target Branch が占める割合を示している。記号実行フェーズで Removable Direction と判断された分岐方向は平均して全体の 30% 程度であった。また、Target Direction と判断された分岐方向は平均して全体の 65% 程度であった。これらに対する分岐予測ミスの抑制やスコアリングの効果は、次のファジングフェーズの評価で行う。

表 6.2: KLEESpectre による解析結果

Program	**GT**	**TP**	**States**	**Speculative States**	**Analysis Time (h:m:s)**	**Max Memory Usage (MB)**
AES	2/1/0	1/1/0	94	378	0:19:37	148.52
ARIA	2/1/1	2/1/1	13	5221	0:1:39	149.74
CAST	1/0/1	1/0/1	8127	507459	0:1:42	83.66
DES	2/2/0	2/2/0	560	21079	5:38:12	4783.49
IDEA	1/1/0	1/1/0	231	66967553	2:54:57	8580.13
MD2	1/1/0	1/1/0	518	13955580	3:39:30	621.91
RC5	1/1/1	1/0/1	41468	40635442	1:32:43	722.34

GT: 埋め込んだ Gadget 数. TP: GT の中で検出された Gadget 数. States: 通常のパスにおける探索された状態数. Speculative States: 投機的なパスにおける探索された状態数

表 6.3: 提案手法の記号実行フェーズによる解析結果

Program	**GT**	**TP**	**States**	**Speculative States**	**Analysis Time (h:m:s)**	**Max Memory Usage (MB)**
AES	2/1/0	1/0/0	91	202	0:19:33	149.19
ARIA	2/1/1	2/0/0	13	533	0:1:08	145.49
CAST	1/0/1	1/0/0	8127	48611	0:1:01	67.89
DES	2/2/0	2/0/0	560	3155	5:27:30	4782.86
IDEA	1/1/0	1/0/0	37740	696804	0:43:20	431.29
MD2	1/1/0	1/0/0	466	28318	0:20:08	199.55
RC5	1/1/1	1/0/0	41468	479269	0:3:43	89.90

表 6.4: KLEESpectre に対する提案手法の記号実行フェーズの解析結果の割合

Program	Speculative States (%)	Analysis Time (%)	Max Memory Usage (%)
AES	53.45	99.66	100.45
ARIA	10.21	68.69	97.16
CAST	9.58	59.80	81.15
DES	14.97	96.84	99.99
IDEA	1.04	24.77	5.03
MD2	0.20	9.17	32.09
RC5	1.28	4.01	14.45

6.4.2 ファジングフェーズの評価

RQ2 に回答するため、提案手法のファジングフェーズを評価した。具体的には、提案手法のファジングフェーズと SpecFuzz を用いてデータセットを解析し、Type2 および Type3 の Gadget の検出率やスループットを比較することで、不要な分岐予測ミスの抑制とスコアによるスケジューリング戦略の効果を検証した。ファジングはシングルスレッドで実行し、各検体に対して 30 分× 4 セット、合計 120 分間実施した結果を比較した。初期シードには、検体の入力が必要とする最低サイズを満たす単一のテキストファイルを用いた。また、提案手法のファジングフェーズで計装に使用する JSON ファイルは、記号実行フェーズの評価時に得られた JSON ファイルを利用した。表 6.5,6.6 に SpecFuzz と提案手法のファジングフェーズによるデータセットの解析結果を示す。また、分岐予測ミスの抑制とスコアによるシードのスケジューリング戦略の効果を評価するために、これらの結果と検体ごとの Removable Branch と Target Branch の割合をまとめたものを表 6.7 に示す。

表 6.7 から、Recall については一部の検体で提案手法が SpecFuzz をわずかに上回る結果が得られたが、Type2 および Type3 の Recall に関しては、提案手法がわずかに劣る傾向が確認された。しかし、この差はファジングという解析手法固有のランダム性に起因するものであり、許容範囲内の誤差と考えられる。したがって、SpecFuzz と提案手法のファジングフェーズには Gadget の検出能力に実質的な差はないと考えられる。

ファジングのスループットに関しては、すべての検体で提案手法が低下しており、平均して約 54% の低下が見られた。また、Removable Direction の割合とスループットの間には明確な正の相関関係は確認されなかった。スループットが低下した理由として、まず SpecFuzz と提案手法のファジングフェーズの投機ウィンドウの計測方法の違いが考えられる。SpecFuzz は LLVM MIR レベルで投機ウィンドウを計測するのに対し、提案手法のファジングフェーズでは KLEESpectre と同じ基準を用いるため、LLVM IR レベルで投機ウィンドウを計測している。この違いにより、同じ投機ウィンドウサイズを設定しても、SpecFuzz の方が早く上限に達してシミュレーションを終了する。このため、SpecFuzz のスループットがより高く出力されたと考えられる。

また、表 6.7 からわかる通り、各検体で平均して全体の 32% 程度の分岐方向の分岐予測ミスしか抑制されていない。加えて、分岐予測ミスの抑制が行われるのは、対象の分岐からシミュレーションが開始される場合に限られる。すでに先行する分岐によってシミュレーションが開始されている場合、ネストされた投機的状態を探索するため、現在の分岐も必ず重ねてシミュレーションが行われる。以上の理由から、分岐予測ミスの抑制が適用される場面は限定的であり、提案手法の分岐予測ミスの抑制やスコア計測を行うための計装に伴う追加の実行コストが、分岐予測ミスの抑制による効果を上回ったことが

スループット低下の原因の一つと考えられる。

次に、スコアを用いたシードのスケジューリング戦略について評価する。提案手法の Recall(Type2,3) は SpecFuzz と比較して向上しなかった。この原因として、ターゲット状態の多さが考えられる。ターゲット状態が多い場合、必然的に Target Direction も多くなる。実際、表 6.6 に示すように、Target Direction の割合は全体的に高く、平均で約 65% を占めている。このような状況では、どのようなテストケースを与えてもスコアが高くなりやすく、有効なテストケースと無効なテストケースを区別することが難しくなる。その結果、特定の状態にテストケースを効果的に誘導することが困難になったと考えられる。今回の実験では、Order 値を 1 に設定したことがターゲット状態は増加に繋がったと考えられる。この設定では、ある条件分岐命令から投機ウィンドウのサイズ（200 命令）以内に別の分岐命令が存在する場合、それがターゲット状態として扱われる。このため、表 6.1 に示す各検体の条件分岐命令数と行数を考慮すると、ターゲット状態の数が非常に多くなってしまうことが予想される。このようにターゲット状態が多い状況では、スコアによる効果的な誘導が難しくなり、提案手法の Recall(Type2,3) が向上しなかったと考えられる。

表 6.5: SpecFuzz による解析結果

Program	**GT**	**30min**	**60min**	**90min**	**120min**	**Recall**	**Recall (Type2,3)**	**Throughput**
AES	2/1/0	1/1/0	1/1/0	1/1/0	1/1/0	0.67	1.00	508
ARIA	2/1/1	2/1/1	2/1/1	2/1/1	2/1/1	1.00	1.00	428
CAST	1/0/1	1/0/0	1/0/0	1/0/0	1/0/0	0.50	0	496
DES	2/2/0	0/1/0	1/1/0	1/1/0	1/1/0	0.50	0.50	496
IDEA	1/1/0	1/1/0	1/1/0	1/1/0	1/1/0	1.00	1.00	380
MD2	1/1/0	0/0/0	1/1/0	1/1/0	1/1/0	1.00	1.00	471
RC5	1/1/1	0/1/1	0/1/1	0/1/1	0/1/1	0.67	1.00	544

Recall: GT のうち TP の割合. Recall(Type2,3): GT を Type2 と Type3 の Gadget に限定した場合の TP の割合. Throughput: ファザーが 1 秒間に実行したテストケースの平均回数.

表 6.6: 提案手法のファジングフェーズによる解析結果

Program	**GT**	**30min**	**60min**	**90min**	**120min**	**Recall**	**Recall (Type2,3)**	**Throughput**
AES	2/1/0	1/1/0	1/1/0	1/1/0	1/1/0	0.67	1.00	227
ARIA	2/1/1	2/1/0	2/1/0	2/1/0	2/1/0	0.75	0.50	278
CAST	1/0/1	1/0/0	1/0/0	1/0/0	1/0/0	0.50	0	307
DES	2/2/0	2/1/0	2/1/0	2/1/0	2/1/0	0.75	0.50	295
IDEA	1/1/0	1/1/0	1/1/0	1/1/0	1/1/0	1.00	1.00	182
MD2	1/1/0	1/1/0	1/1/0	1/1/0	1/1/0	1.00	1.00	188
RC5	1/1/1	1/1/1	1/1/1	1/1/1	1/1/1	1.00	1.00	308

以上の結果を踏まえ、ターゲット状態への効率的な誘導を実現するために、以下の 2 つの方法が考えられる。1 つ目は、Order 値を増加させて解析を行う方法である。今回の実験では Order 値を 1 に設定したため、ターゲット状態が多くなったが、Order 値を上げることでファジングフェーズで探索するターゲット状態数を減らし、スコアによる評価がより効果的に機能すると考えられる。ただし、この方法では、Order 値を増加させることで記号実行フェーズにおける投機的状態の探索が増加し、記号実行

表 6.7: 提案手法のファジングフェーズと SpecFuzz の解析結果の比較

Program	Removable Directions(%)	Target Directions(%)	SpecFuzz		提案手法		Throughput (%)
			Recall	Recall (Type2,3)	Recall	Recall (Type2,3)	
AES	38.00	36.00	0.67	1.00	0.67	1.00	44.69
ARIA	47.22	68.08	1.00	1.00	0.75	0.50	64.95
CAST	28.13	71.19	0.50	0	0.50	0	61.90
DES	35.71	60.71	0.50	0.50	0.75	0.50	59.48
IDEA	40.91	61.36	1.00	1.00	1.00	1.00	47.89
MD2	9.09	72.27	1.00	1.00	1.00	1.00	39.92
RC5	23.81	84.52	0.67	1.00	1.00	1.00	56.62

Removable Directions (%): 全ての分岐方向に対する Removable Direction の割合. Target Directions (%): 分岐予測ミスを含む全ての分岐方向に対する Target Direction の割合. Throughput (%): SpecFuzz の Throughput に対する提案手法のファジングフェーズの Throughput の割合.

のスケーラビリティが低下する可能性がある。したがって、検体ごとに記号実行フェーズとファジングフェーズの効果を最大限に引き出す最適な Order 値を設定する必要があると考える。

2 つ目は、より詳細なスコアリング手法を導入する方法である。本研究で提案した手法では、テストケースが通過した Target Direction の種類数が多いほどスコアが高くなるという単純なスコアリング手法を採用している。そのため、この手法にはテストケースが通過した分岐方向の順序や複数のターゲット状態を区別する要素が含まれていない。その結果、ターゲット状態が多い場合、どのようなテストケースでも一定のスコアを獲得しやすくなり、テストケースの有効性を細かく評価できていなかったと考えられる。分岐方向の通過順序やターゲット状態ごとのスコアを導入することで、テストケースの有効性をより詳細に評価できる可能性がある。しかし、詳細なスコアリングを行うためには、より複雑な計装が必要となるため、その分実行時のオーバーヘッドが増加し、ファジングのスループットが低下する可能性がある。このため、実行時オーバーヘッドの増加を最小限に抑えつつ、効率的にテストケースを誘導できる手法を検討する必要があると考える。

6.4.3 提案手法全体の評価

RQ3 に答えるため、提案手法全体を KLEESpectre および SpecFuzz と比較しその性能を評価した。本研究では、各ツールの効率性を図る指標として、単位時間あたりの Gadget 検出数およびメモリ使用量あたりの Gadget 検出数を使用する。つまり、短い解析時間または少ないメモリ使用量で多くの Gadget を検出できたツールを効率的であると評価する。

使用したデータは、RQ1 および RQ2 で行った実験の結果を利用したものである。具体的には、ファジングによる解析では 120 分間の実行で検出された Gadget を基に、記号実行による解析では解析が終了するまでに要した時間を基に、それぞれ単位時間あたりの Gadget 検出数を計算した。一方、提案手法では記号実行フェーズの解析に要した時間と、ファジングフェーズでの解析時間 (120 分) の合計を全体の解析時間とし、単位時間あたりに検出された Gadget 数を計算した。メモリ使用量あたりの Gadget 検出数は、各解析ツールによる解析が終了するまでに必要とした最大メモリ使用量を用いて計算を行った。

表 6.8 に、各解析ツールについて検体ごとの単位時間あたりの Gadget 検出数およびメモリ使用量あたりの Gadget 検出数を示す。表 6.8 が示す通り、提案手法は RC5 の検体において、KLEESpectre および SpecFuzz を上回る単位時間あたりの Gadget 検出数を記録した。しかし、それ以外の検体では提案手法の Gadget 検出数が他のツールより低下している。RC5 において提案手法がより優れた値を記

表 6.8: 提案手法とその他の解析ツールとの比較

Program	KLEESpectre		SpecFuzz		提案手法	
	Gadget/m	Gadget/MB	Gadget/m	Gadget/MB	Gadget/m	Gadget/MB
AES	0.103	0.0135	0.017	0.1362	0.022	0.0201
ARIA	2.878	0.0277	0.033	0.2712	0.025	0.0206
CAST	1.408	0.0239	0.008	0.0669	0.008	0.0147
DES	0.012	0.0008	0.017	0.1350	0.007	0.0006
IDEA	0.011	0.0002	0.017	0.1359	0.012	0.0046
MD2	0.009	0.0032	0.017	0.1221	0.014	0.0100
RC5	0.022	0.0028	0.017	0.1359	0.024	0.0334

Gadget/m: 1 分間あたりに検出した Gadget 数. Gadget/MB: メモリ使用量 (MB) あたりに検出した Gadget 数.

録した理由として、Order 制限を加えた記号実行によって KLEESpectre よりも大幅に解析時間が短縮したことと、Type1 の Gadget が特定の条件下でのみ到達可能なパスに存在しており、SpecFuzz では見逃されたことが挙げられる。

メモリ使用量あたりの Gadget 検出数では全ての検体において、SpecFuzz が最も優れていた。しかし、提案手法は 4 つの検体で KLEESpectre よりもメモリ使用量あたりの Gadget 検出数が多かった。この理由として、Order 制限により記号実行のメモリ使用量が抑えられたことと、ファジングによる解析が記号実行よりも少ないメモリ使用量で一部の Gadget を検出できていたことが挙げられる。提案手法では、SpecFuzz ほどメモリ効率が良く Gadget を検出できなかったものの、最大メモリ使用量はいずれの検体においても 5GB 以下に収まった。また、KLEESpectre と比較すると、平均して最大メモリ使用量が約 61% 削減された。このことから、提案手法はメモリリソースが限られた環境でも多くの検体を十分に解析可能であると考える。

以上の結果から、Order 制限により投機的状態数が大幅に抑制されるとともに、Type1 の Gadget が特定の条件下でのみ到達可能なパスに多く含まれる検体においては、提案手法がスケーラビリティと精度の両立を実現できる可能性があると考える。

第 7 章

今後の展望

7.1 最適な Order 値の模索

今回の提案手法の評価では、Order 値を 1 に設定して実験を行った。この設定により、記号実行フェーズで探索される投機的状態数は減少したが、その分ファジングフェーズで探索するべき投機的状態数が増加した。ファジングのターゲット状態が増加すると、ファジング特有のランダム性の影響で、検体に Type2 や Type3 の Gadget が多く含まれる場合に false negative が増加する可能性がある。一方で、Order 値を上げて記号実行フェーズで探索する投機的状態数を増やすと、記号実行のスケーラビリティが低下する問題が生じる。よって今後の研究では、このトレードオフを考慮して、より多くの検体に対して様々な Order 値を設定し詳細な実験を行うことで、スケーラビリティと精度の両立を実現する最適な Order 値を特定する必要があると考えられる。

7.2 スコアリング手法の改善

提案手法では、テストケースが通過した Target Direction の種類数が多いほどスコアが高くなるという単純なスコアリング手法を採用している。しかし、予備実験の結果から、この手法では十分に効果的にテストケースをターゲット状態に誘導できていないことが示唆された。これを改善するためには、より詳細な情報を活用してテストケースを評価する手法が必要であると考える。例えば、分岐方向の通過順序を考慮したり、ターゲット状態ごとにスコアを個別に計測することで、テストケースの有効性をより正確に評価できる可能性がある。また、一般的な directed fuzzer [8] のアプローチを参考にし、制御フローグラフ上でターゲット状態までの距離を考慮する手法も有効と考えられる。ただし、提案手法では、ターゲット状態は投機的な状態であるため、通常の制御フローグラフ上には存在しない。したがって、投機的なパスを含めた制御フローグラフを構築する必要があると考えられる。いずれにしても、スコアリング手法が複雑化すると、それに伴いファジングのスループットが低下する可能性があるため、効率的な設計が必要となる。

7.3 テストドライバの設計

記号実行やファジングを用いてプログラムを適切にテストするには、テストドライバが必要不可欠である。テストドライバでは、記号実行において記号化する変数を指定したり、ファジングにおいて生成されたシードを用いてプログラムの入力を正しく初期化する役割を担う。また、これらの変数を対象の関数などに渡して実行する。テストドライバの設計は解析の精度やスケーラビリティに大きく影響する。例えば、記号実行の場合、適切に変数を記号化しなければ特定のパスを探索できない可能性がある。一方で、すべての変数を記号化すると状態爆発が発生し、スケーラビリティが著しく低下するリス

クがある。同様に、ファジングでは、変数が正しく初期化されていない場合、早期にエラーが発生してカバレッジが向上しない可能性がある。本研究の実験では、すべての検体に対して自作のテストドライバを使用した。そのため、解析の精度やスケーラビリティがこれらのドライバの設計に依存していた可能性がある。記号実行においては、制御フローに影響するすべての変数を特定し記号化することで、全てのパスを探索できると考える。ファジングでは Fudge [6] のようなファジングドライバを自動生成するツールを活用することで、適切なドライバを用いた実験が可能となると考える。

7.4 実世界のプログラムにおける Gadget の調査

本研究の実験では、実世界のプログラムにおける ground truth が限られているため、他のツールとの定量的な比較が困難であった。この課題に対処するため、既知の Gadget を埋め込んだデータセットを作成し、実験を行った。具体的には、提案手法の有効性を評価するため、検体全体にわたって Type1 の Gadget とそれ以外の Gadget を均等に 10 個ずつ埋め込んだ。しかし、実世界のプログラムにおいて各 Type の Gadget がどのような割合で存在しているかは明らかではない。先行研究 [30] の調査では、一部の検体では Type1 の Gadget が最も多く含まれていることが示されているが、この割合によっては適切な Order 値が変化する可能性があるためより広範囲な調査が必要である。そのため、多くの実世界のプログラムを対象に各 Type の Gadget の割合を調査することが、今後の研究において重要であると考えられる。

第 8 章

関連研究

8.1 Transient Execution Attack

transient execution attack は、大きく分けて Spectre 型の攻撃と Meltdown 型の攻撃に分類される。Spectre 型の攻撃は Spectre-PHT [22]、Spectre-BTB [22]、Spectre-RSB [24, 27]、Spectre-STL [15] の 4 つに分類され、いずれもキャッシュをサイドチャネルとして利用することに重点を置いている。Spectre-PHT 攻撃は CPU の Pattern History Table (PHT) を操作することで誤った投機実行を誘発させる。Spectre-BTB 攻撃は間接ジャンプの飛び先を予測するための機構である Branch Target Buffer (BTB) を操作することで、制御フローを操作して攻撃を行う。Spectre-RSB 攻撃はリターンアドレスを予測するための機構である Return Stack Buffer (RSB) を操作することで、制御フローを操作して攻撃を行う。Spectre-STL 攻撃は先行するストア命令が全て完了する前にロード命令される値を予測して利用する Store-To-Load forwarding (STL) による投機実行を悪用する攻撃である。

Meltdown 型の攻撃は、CPU のアウト・オブ・オーダー実行によって、例外発生後に一時的に実行される命令を悪用する攻撃である。例えば、ユーザー空間からカーネルメモリの読み取り [25, 40, 46] や無効なレジスタや特権レジスタの読み取り [19, 37] などによって発生する例外を悪用することで攻撃を行う。

Meltdown 型の攻撃に対しては、ハードウェアレベルの対策 [18] やカーネルによる対策 [12] など効率的な防御策が提供されている。一方、Spectre 型の攻撃に対しては、ハードウェアレベルでの対策は未だ提供されておらず、ソフトウェアレベルでパフォーマンスを低下させず完全に防御することは困難である。本研究は Spectre-PHT 攻撃に悪用される Spectre Gadget の検出のみに焦点を当てている。

8.2 Spectre Gadget の検出

Spectre 攻撃に対する最も保守的な防御法は全ての投機実行を無効にすることである。例えば、Spectre-PHT 攻撃への対策として、全ての分岐命令やメモリアクセス命令に対して直列化命令や SLH [11] を適用することができる。しかし、既存研究 [30, 44] が示すように、この方法は非常に大きな実行時オーバヘッドを伴う。そこで、プログラム中の Spectre 脆弱性を検出することで、全ての命令に対して防御を適用することを避けるために、Spectre Gadget 検出ツールが利用される。

Spectre Gadget の検出手法は、静的解析手法と動的解析手法に分類できる。静的解析手法では、Spectre 1 Scanner [4] のように構文パターンを用いて Spectre Gadget をモデル化し、バイナリに対して静的なパターンマッチングを行う方法がある。この手法は、定義済みのパターンにのみマッチする Gadget を検出するため、見逃しが発生する可能性があり、また単純なパターンマッチングに基づいているため、多くの誤検出が生じる。一方、oo7 [44] では、静的テイント解析を利用し、Spectre Gadget 特有のデータフローを捉えることで検出を行う。この方法は構文パターンに基づく手法よりも正確に

Spectre Gadget をモデル化しているが、静的解析に特有の保守的な解析によって過剰汚染が発生し、多数の誤検出を引き起こす。SPECTECTOR [14] と KLEESpectre [43] は記号実行を使用して Spectre Gadget を検出する。これは他のパス非依存な静的解析手法よりも正確であり、動的解析手法と比較してプログラムを網羅的に解析することができる。しかし、記号実行に固有の問題である状態数の爆発により、大規模な検体に対してはスケールしないという問題がある。この問題は特に、投機的なパスを考慮する必要がある Spectre Gadget の検出において顕著である。それを踏まえて、本研究では、記号実行においてネストされた分岐予測ミスのシミュレーションに制限をかけることで、状態数の爆発を軽減し、スケーラビリティを向上させる手法を提案した。

動的解析手法としては、ファジングやテイント解析を活用した検出手法が提案されている。その中でも SpecFuzz [30] は、対象プログラムに計装を施し、投機実行をシミュレートすることで、ファジングを用いてシミュレーション中に発生した境界外アクセスを検出する。SpecFuzz は、ネストされた分岐予測ミスによって悪用可能な Gadget が少ないという観測に基づき、提案手法の記号実行フェーズと同様に、ネストされた分岐予測ミスのシミュレーションを抑制することでファジングのスループットを向上させている。しかし、このアプローチにより、複数回の分岐予測ミスを必要とする Gadget は見逃される可能性が高い。ファジングを利用した手法は記号実行に比べて大規模なプログラムにもスケールしやすいという利点があるが、カバレッジが不足すると脆弱性を見逃す可能性がある。また、SpecFuzz は単純な検出ロジックおよびシミュレーションロジックを採用しているため、誤検出が多発する可能性がある。

SpecTaint [31] は動的テイント解析を利用した手法で、攻撃者が操作可能な値を汚染することで、実際に悪用可能な Gadget のみを検出する。また、SpecTaint は例外を考慮した投機実行のシミュレーションを行うことで、より実際の CPU に近い動作を再現し、精度の高い解析を実現している。しかし、SpecFuzz と比較してシミュレーションロジックの複雑化によりファジングのスループットは低下している。KASPER [20] は Linux カーネルを対象としたファジングによる Gadget 検出ツールであり、transient execution attack で使用される手順をモデル化することで既存のツールより幅広い脆弱性を検出する手法を提案した。KASPER は本研究の記号実行フェーズと同様に、カーネルのような大規模な検体に対してのスケーラビリティを向上させるために、ネストされた分岐予測ミスのシミュレーションを制限している。その結果、ネストされた分岐予測ミスにより悪用可能な Gadget は見逃されることになる。提案手法はファザーをネストされた分岐予測ミスにより到達する投機的な状態を探索するように誘導することで、そのような Gadget の検出を行う。

これらの動的解析手法に共通することは、ファジング固有の問題であるカバレッジ不足により脆弱性が見逃される可能性がある点である。本研究では、ファジングを記号実行と組み合わせることで、スケーラビリティを維持しつつ、より高い精度で Spectre Gadget を検出する手法を提案した。

8.3 ハードウェアによる防御法

Spectre 攻撃に対する根本的な防御策として、ハードウェアに変更を加えるアプローチが挙げられる。InvisiSpec [47] は、安全でない投機実行中のロード命令がキャッシュではなく、専用のハードウェア機構である投機バッファにデータをロードする手法を提案している。投機実行が正しかった場合には、投機バッファの内容がキャッシュに反映され、誤りだった場合にはその内容が破棄される。この手法により、投機実行中のロード命令がキャッシュの状態に影響を与えなくなるため、キャッシュをサイドチャネルとして悪用することができなくなる。しかし、このアプローチには、予測が正しかった場合でも投機バッファへのロードとキャッシュへの反映という二重のコストが伴うという課題がある。

CleanupSpec [33] は、予測ミスが検出された際に、一時的な命令によってキャッシュに加えられた不正な変更をロールバックする手法を提案している。このアプローチは InvisiSpec とは異なり、正しい予

測が行われた場合のロード命令は通常通りキャッシュにアクセスするだけで済むため、比較的オーバーヘッドは小さい。ただし、予測ミスが頻発する場合はキャッシュ状態のロールバックにかかるオーバーヘッドが非常に大きくなると考えられる。

NDA [45] は、ハードウェアレベルで投機実行中のデータ伝播を制限する手法を提案している。この手法では、ソース命令が安全であると判断されるまで、その命令の結果を依存先に伝播させることを遅延させることで、誤った投機実行によるデータ漏洩を防ぐ。ConTExT [34] は、CPU のページテーブルエントリビットを拡張し、一時実行中のアクセスを制限する秘密情報が含まれていることを示す手法を提案している。この拡張により、CPU が投機実行中にこれらの秘密情報にアクセスしようとした場合、実際のデータの代わりにダミー値を使用することで、秘密情報がマイクロアーキテクチャの状態に漏洩することを防ぐ。ただし、この手法を利用するには、ユーザがソースコードレベルで秘密情報にアノテーションを付与することと、秘密情報を追跡するための動的テイント解析を実行することが必要である。

ハードウェアレベルの防御手法は、Spectre 攻撃の原因に直接対処することで、ソフトウェアレベルの防御手法に比べてパフォーマンスの低下を最小限に抑えることができると考えられる。しかし、ハードウェア設計が複雑化することや、既に市場に出回っている CPU にこれらの防御手法を適用できないといった問題も存在する。

8.4 その他の防御法

ここでは、前述の手法以外の Spectre 攻撃に対する防御手法について述べる。Site Isolation [32] は、ブラウザにおける防御策であり、各ウェブサイトをそれぞれサンドボックス化されたプロセスに分離することで、信頼できない JavaScript コードが同一プロセス内で実行されるのを防ぐ。これにより、攻撃者と被害者が同じメモリ空間を共有しないため、transient execution attack の難易度が向上する。しかし、この手法ではプロセス数が大幅に増加するため、メモリオーバーヘッドを低減するための最適化が必要となる。

SpecCFI [23] は、制御フローハイジャック攻撃を防御するための手法である Control-Flow Integrity (CFI) を活用し、Spectre 攻撃（Spectre-BTB および Spectre-RSB）を防ぐ手法を提案している。この手法では、CFI を利用して間接分岐の可能なターゲットを制御フローグラフによって事前に定義された正当なターゲットに制限し、投機実行による不正な制御フローを抑制する。CFI はすでに一般的なハードウェアで利用可能であるため (Intel の CET や ARM の BTI など)、これらを活用することで、ハードウェア拡張を最小限に抑えつつ、小さいオーバーヘッドで Spectre 攻撃を防御をする手法を提案している。

Swivel [29] は、WebAssembly（Wasm）のサンドボックス環境を Spectre 攻撃に対して強化する手法を提案している。Wasm コンパイラは、プログラムをコンパイルする際に、制御フローをサンドボックス内に制限し、全てのメモリアクセスをサンドボックス内に収めるようマスク処理を施す。これにより、外部のサンドボックスへのアクセスを防止する。しかし、投機実行中にこれらの Wasm の分離保証がバイパスされる可能性が依然として存在するため、Spectre 攻撃が可能である。この問題に対処するため、Swivel はプログラムをリニアブロックと呼ばれる、終端以外に制御フロー命令を含まない直線的なコードブロックに分割する。さらに、コンパイル時に、逐次的または投機的な実行に関係なく、リニアブロックの先頭にのみジャンプするように制約を設ける。この手法により、投機実行中にメモリアクセスのマスク処理がバイパスされることを防ぐ。

第 9 章

おわりに

本研究では、記号実行とファジングの解析手法を組み合わせることで、スケーラビリティと精度の両立を目指す Spectre Gadget の検出手法を提案した。記号実行とファジングにはそれぞれ利点と欠点が存在するが、両者を組み合わせることで、それぞれの欠点を補完し合い、スケーラビリティと精度を両立させることを目的とした。記号実行フェーズでは、ネストされた分岐予測ミスの回数を制限することで、投機的状態の探索範囲を縮小し、記号実行のスケーラビリティを向上させる手法を用いた。また、ファジングフェーズでは記号実行で得られた解析結果を活用し、記号実行で探索されなかった状態を効率的に探索することを試みた。予備実験では、いくつかの検体において記号実行のスケーラビリティが大幅に向上することを確認した。さらに、提案手法全体としては、既存手法よりも効率的に Spectre Gadget を検出した検体を確認した。今後の展望として、スケーラビリティと精度の向上を図るため、最適な Order 値の探索やスコアリング手法の改善が必要であると考える。これにより、提案手法がより多くの検体に対して有効に機能するようになると考える。

謝辞

本論文の執筆にあたり、権藤克彦教授および荒堀喜貴助教授には、日頃からゼミにおける指導や助言をいただき、大変お世話になりました。また、ゼミを通じて研究室の皆様から多くの知見や刺激をいただき、大変励みになりました。特に、同期である山本君と上野君とは研究について多くの議論を交わし、貴重な洞察を得ることができました。本研究を支えてくださったすべての皆様に、この場を借りて感謝を申し上げます。

参考文献

[1] Honggfuzz. https://honggfuzz.dev/ Accessed: January 2025.

[2] Openssl. https://openssl-library.org/ Accessed: January 2025.

[3] Refined speculative execution terminology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html.

[4] Spectre variant 1 scanning tool, 2018. https://access.redhat.com/blogs/766093/posts/3510331.

[5] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In Information and Communications Security: 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006. Proceedings 8, pages 112–121. Springer, 2006.

[6] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 975–985, 2019.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D' elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 51(3):1–39, 2018.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pages 2329–2344, 2017.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In 28th USENIX Security Symposium (USENIX Security 19), pages 249–266, 2019.

[11] Chandler Carruth. Speculative load hardening, 2018. https://llvm.org/docs/SpeculativeLoadHardening.html.

[12] Jonathan Corbet. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/ Accessed: January 2025.

[13] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. Software optimization resources, 2016.

[14] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1–19. IEEE, 2020.

[15] J. Horn. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 Accessed: January, 2025.

[16] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018.

[17] INTEL. Affected processors: Guidance for security issues on intel® processors. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html#tab-blade-1-1 Accessed: January 2025.

[18] INTEL. L1 terminal fault / cve-2018-3615 , cve-2018-3620,cve-2018-3646 / intel-sa-00161. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html Accessed: January 2025.

[19] INTEL. Q2 2018 speculative execution side channel update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html Accessed: January 2025.

[20] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the linux kernel. In NDSS, volume 1, page 12, 2022.

[21] Paul Kocher. Spectre mitigations in microsoft's c/c++ compiler, 2018. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html Accessed: January 2025.

[22] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1–19, 2019.

[23] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In 2020 IEEE Symposium on Security and Privacy (SP), pages 39–53, 2020.

[24] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In 12th USENIX Workshop on Offensive Technologies (WOOT 18), Baltimore, MD, August 2018. USENIX Association.

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[26] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In 2015 IEEE symposium on security and privacy, pages 605–622. IEEE, 2015.

[27] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. CCS '18, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and

mitigations. In Proceedings of the 35th Annual Computer Security Applications Conference, pages 747–761, 2019.

[29] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening {WebAssembly} against spectre. In 30th USENIX Security Symposium (USENIX Security 21), pages 1433–1450, 2021.

[30] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In 29th USENIX Security Symposium (USENIX Security 20), pages 1481–1498, 2020.

[31] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In NDSS, pages 1–14, 2021.

[32] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In 28th USENIX Security Symposium (USENIX Security 19), pages 1661–1678, 2019.

[33] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An" undo" approach to safe speculation. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 73–86, 2019.

[34] Michael Schwarz, Moritz Lipp, Claudio Alberto Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. In Network and Distributed System Security Symposium 2020, 2020.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In 2012 USENIX annual technical conference (USENIX ATC 12), pages 309–318, 2012.

[36] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on {SSH}. In 10th USENIX Security Symposium (USENIX Security 01), 2001.

[37] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. arXiv preprint arXiv:1806.07480, 2018.

[38] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. Journal of Hardware and Systems Security, 3(3):219–234, 2019.

[39] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. Journal of Cryptology, 23:37–71, 2010.

[40] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In 27th USENIX Security Symposium (USENIX Security 18), pages 991–1008, 2018.

[41] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In 2020 IEEE Symposium on Security and Privacy (SP), pages 54–72. IEEE, 2020.

[42] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In 2019 IEEE Symposium on Security and Privacy (SP), pages 88–105. IEEE, 2019.

[43] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roy-

choudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. ACM Transactions on Software Engineering and Methodology (TOSEM), 29(3):1–31, 2020.

[44] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. arXiv preprint arXiv:1807.05843, 2018.

[45] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 572–586, 2019.

[46] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[47] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 428–441. IEEE, 2018.

[48] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In 23rd USENIX security symposium (USENIX security 14), pages 719–732, 2014.